INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Monolítico Modular: Análise do desempenho, da manutenibilidade e da eficiência energética

NUNO JOSÉ ÁLVARO MARMELEIRO
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Modular Monolithic: Performance, maintainability and energy efficiency analysis

Nuno José Álvaro Marmeleiro

Dissertation

Master's in computer engineering

Specialisation in Software Engineering

Supervisor: Isabel de Fátima Silva Azevedo

Porto, June 2025

Statement of Integrity

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section “Ethical considerations” of the first chapter. This section also states how AI tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, 29 June 2025

Nuno José Álvaro Marmeleiro

Abstract

This study investigates how modularity affects software maintainability, performance, and energy efficiency, supporting modular monolithic design as a middle ground step between the migration from monolithic design to a microservice design. A controlled experiment was conducted, comparing the values from several metrics of maintainability, performance and energy efficiency in applications featuring varying degrees of modularity. This work has results that show how different levels of modularity affect software, such as increasing maintainability and performance, but having more energy consumption. This paper gives evidence for the modular designs and their subjective contribution to maintainability and performance, but also the trade-offs in terms of energy efficiency. This work helps in the architectural structure decision process, providing information for developers and organisations which desire to transition from non-modular to modular architectures. Future work includes extending the analysis to additional request types, quality attributes, more application contexts, and larger software systems.

Keywords: Modularity, Monolithic Architecture, Modular Monolithic, Software Quality, Maintainability, Performance, Energy Efficiency

Resumo

Este estudo investiga a forma como a modularidade afeta atributos de qualidade do software, como a manutenção, o desempenho e a eficiência energética, apoiando a modularidade monolítica como um passo intermédio na migração de uma aplicação monolítica para uma aplicação de microsserviços. Foi realizada uma revisão de literatura, que inclui informação sobre tópicos já pesquisados e investigados na área, indicando que a modularidade melhora o desempenho, contudo, em certos contextos, este pode piorar devido à sobrecarga introduzida pela implementação de técnicas modular. O mesmo se verifica em relação à manutenabilidade, devido ao aumento de componentes introduzidos numa aplicação desta arquitetura. A nível de consumo energético, também se verifica uma tendência negativa devido à intercomunicação adicional entre componentes. Para obter resultados válidos e transparentes, foi realizada uma experiência controlada, selecionando uma aplicação com base em vários fatores, seguida de uma análise e comparação de vários valores de métricas de manutenção, desempenho e eficiência energética em aplicações com diferentes graus de modularidade. Os resultados obtios mostram que o aumento da modularidade melhora a manutenção e o desempenho, em contra partida, aumenta o consumo energético. Esta experiência, embora mostre que a modularidade pode trazer certos compromissos ligados à eficiência energia, apresenta evidências de que a modularidade tem um efeito positivo na manutenção e no desempenho. Este estudo contribui com conhecimentos para apoiar decisões nessa escolha da arquitetura de aplicações de software, fornecendo informações importantes aos programadores e às empresas que pretendam migrar de arquiteturas não modulares para arquiteturas modulares. O trabalho futuro inclui o alargamento da análise a mais tipos de pedidos, a outros atributos de qualidade, a mais contextos de aplicação e a sistemas de software de maior dimensão.

Palavras-chave: Modularidade, Aplicações Monolíticas, Qualidade de Software, Escalabilidade, Manutenibilidade, Desempenho.

Acknowledgment

First and foremost, I would like to thank myself for the persistence and dedication it took to complete this journey and bring this work into existence.

I want to thank my supervisor, who shared valuable knowledge with me and helped me achieve higher results.

I am also deeply grateful to my friends and family for their support and encouragement throughout this project. They were a source of constant motivation because they believed in me.

I would also like to thank my university colleagues, who were in the same position as I, going through the same challenges and milestones. Their insights and different perspectives were most useful in bringing my thinking to new levels.

Finally, I would like to make mention of my very best friend Joana, with a deeply emotional thank you. She was the person who most concerned me at this point, standing by me for better or for worse. Her presence, guidance, and encouragement were crucial to the success of this dissertation. Without her, this paper would most likely never have existed.

Table of Contents

List of Figures

List of Tables

List of Code Snippets

List of Abbreviations

ACD	Average Component Dependency
AMQP	Advanced Message Queuing Protocol
CBO	Coupling Between Objects
CCD	Cumulative Component Dependency
CPU	Control Process Unit
GQM	Goal/Question/Metric
IoT	Internet of Things
LOC	Lines Of Code
MMI	Modularity Maturity Index
NCCD	Normalised Cumulative Component Dependency
NOC	Number Of Classes
NOM	Number Of Methods
PRISMA	Preferred Reporting Items for Systematic reviews and Meta-Analyses
UI	User Interface
VM	Virtual Machine
WBS	Work Breakdown Structure

1 Introduction

This chapter introduces the context of this project and the problem in the section 1.1, followed by its objectives and contributions, in the section 1.2 and section 1.3. Finally, it takes into consideration the ethical considerations and the structure of the document is briefly explained, in sections 1.4 and 1.5.

1.1 Context and Problem

Many microservices-based applications can be seen as distributed monolithic with “all the disadvantages of a distributed system, and the disadvantages of a single-process monolithic, without having enough of the upsides of either” [1]. Dealing with additional complexity might not bring the foreseen benefits. The Istio open-source service mesh embraced a microservice architecture to reverse that decision later [2]. Companies tend to move from monolithic to microservices to reach higher maintainability [3], but this comes with drawbacks. The rollback from microservice to monolithic is not made because of the modularity increase, since it helps the maintainability, but because of the complexity, cost and performance decrease that microservices can offer [4]. Thus, a modular monolithic architecture benefits in these fields, having less complexity, less cost, and even adding the chance to learn about the modular design before doing the final migration to microservices when needed [4].

Architectural modularity does not dictate a distributed architecture [5]. Modular monolithic is the choice of many organisations, such as Shopify, which explained in their blog [6] and held a conference about it [4], Istio [2], Amazon, in their Video Monitoring service, Segment and InVision [4]. If the module boundaries are well-defined, achieving a high degree of parallel work is possible [5] and the benefits that modularity brings can still be achieved [4].

Modularity, directly and indirectly, affects other quality attributes [5]. Modularity adds agility, a compound characteristic evolving maintainability, testability and deployability, to the application, supporting speed-to-market by enabling the system to respond quickly to change [5]. Modularity also brings maintainability to the application, facilitating the addition, change or removal of functionalities [5].

However, not all effects are positive. Modularity can bring some trade-offs to an application. Test complexity is one of them. While modularity can improve testability by isolating test scopes, excessive service interdependencies can lead to declining testability.

Modularity also indirectly influences fault tolerance, allowing the faults to be contained within isolated components, enabling the rest of the system to function normally. However, synchronous dependencies can compromise this factor if one failing service affects the others [5].

Even before the term modular monolithic emerged, the focus of developing a monolithic architecture was already on modularity [6]. Yet, some companies keep underestimating this step [2].

Even though the modular monolithic design has started to gain popularity, and the idea is growing, however, some people still have doubts about this architectural design [7].

1.2 Objectives

This work explores the impact of modularity on maintainability, performance, and energy efficiency in applications, as well as the analysis of their architecture and relevance to modern software development. By analysing their bases, origins, and utilisation, this research aims to explore the advantages and disadvantages of this architecture design in contrast to non-modular monolithic and microservices architectures.

This includes an in-depth examination of how modularity influences maintainability, performance, and energy efficiency, both negatively and positively, in a controlled experience using a single application based on specific factors, ensuring the same functionalities for transparent results. Given the limited timeframe to develop this research, migrate applications, and perform the evaluation, only the three software quality attributes mentioned previously will be considered.

Modularity is a sub-characteristic of maintainability, which includes other sub-characteristics - reusability, analysability, modifiability and testability, that ultimately are also in the analysis [8]. Energy efficiency has become a relevant topic to take into consideration in developing software [9] and performance has a highly significant aspect on the services and goods to the customers [10].

The study offers a broad perspective on the real-world uses of modular monolithics in response to scepticism in related industries. It assesses their ability to drive system design and reduce the challenges of other architectural models. With this study, the authors hope to complement the growing literature on modular monolithic design and its strategic role in achieving efficient, sustainable, and maintainable software systems.

1.3 Contributions

The advantages of this study are that it is far-reaching and addresses academic, industry, technical, and sustainability perspectives. From an academic point of view, this study shows results that contribute to the knowledge of software modularity and architecture, supporting further research in software methodologies [11]. From an industry point of view, this study helps companies to make better decisions when it comes to migration strategies and modularity adoption, which are key concerns for companies that want to achieve better maintainability and performance [1]. On the technical side, this study also analyses the practical implications

that are associated with some architectural designs, showing trade-offs between modularity, performance and energy efficiency to software developers [12]. Finally, from the point of view of sustainability, this study shows the impact of software architectures on the environment, which are sometimes overlooked, by showing the values of energy efficiency and consumption metrics, contributing to greener and more sustainable software engineering practices [13].

The research contributes to knowledge of modular monolithic, an architecture method that is gaining acceptance but not yet widely investigated, and the impact of modularity in software systems. By analysing the trade-offs between modular monolithic, non-modular monolithic, and microservices, this study offers insightful information that closes current knowledge gaps.

By providing practitioners with up-to-date information on the subject, this study allows organisations to select from a variety of architectural approaches. It clarifies the useful applications of modularity, especially regarding energy efficiency and maintainability, two important factors in contemporary software development.

This method gives the industry the justification it needs to lessen the frequency of reversing design decisions by clearly illustrating the advantages and disadvantages of modularising software applications [4]. It promotes informed decision-making, enabling high-value, sustainable solutions to be created.

1.4 Ethical Considerations

Ethical considerations are very important in software engineering research, affecting the research process's integrity and transparency. This section highlights the ethical standards used to develop this work.

This study has been conducted strictly with the ethical guidelines and principles laid out by the Ordem dos Engenheiros [14] and the IEEE Code of Ethics [15]. Each stage of this project reflects a commitment to these standards, ensuring responsible, transparent, and accountable research practices. Ethical considerations also obligate researchers to recognise limits to the work done and mention worthy future work needed about the topic discussed, which is considered.

Additionally, the Declaration of Integrity included in this document fulfils the requirements outlined in the Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto [16], demonstrating our dedication to maintaining high ethical standards throughout the project. By fully adhering to these guidelines, this work upholds the principles of fairness, responsibility, and respect for both the profession and the broader community.

Grammarly was used throughout the writing of this document purely for grammar and sentence correction—no generative artificial intelligence tools were involved in this process. Generative artificial intelligence tools were only used to help understand how certain software tools work or to speed up the creation of small code snippets. Even so, all generated content was carefully reviewed and verified by the author.

1.5 Document Structure

This document contains seven chapters. These include Introduction, Skills Management and Project Planning, Background, Literature Review, Experiment, Results and Conclusion:

- Introduction: The current chapter explains the context and problem, objectives, contributions of this project, ethical considerations and the document structure.

- Skills Management and Project Planning: This chapter references the skills needed and the plan of action taken into consideration to develop this work. Additionally, the project planning is also mentioned with the necessary steps taken to achieve the success of this work.

- Background: Explains the key concepts to understand the literature review of this paper and the work developed: Software Quality, explaining the software quality attributes modularity, performance and energy efficiency, the understanding of what a monolithic application and microservices, what code smells represent in software development and the event-driven design concept.

- Literature Review: A crucial chapter of this paper that resumes the research already developed on the effects of modularity in applications and the reason why companies in this field aim to migrate to a microservice architecture.

- Experiment: This chapter delves into the application criteria selection, talking about several factors that contributed to the choice of application and the project chosen to evaluate, explaining its business context and the architecture, for the three application types: non-modular monolithic, modular monolithic and microservices. Further, it also explains the process of certain adjustments and migrations needed to transform the applications into the most similar, so that all the functionalities and class structures are the same, to reduce threats to validity concerning these points.

- Results: Focus on the explanation of the methodology of evaluation, based on Goal-Question-Metric, followed by the experiments performed on performance, maintainability and energy efficiency.

- Conclusion: A chapter to conclude the paper, which includes the achievements and contributions, difficulties, threats to validity, future work and the final considerations.

2 Skills Management and Project Planning

In this chapter, the skills needed to develop this document are referenced in Section 2.1, followed by the planning of the project in Section 2.2. It aligns with the Software Engineering Code of Ethics, on point 1.03, “Ensure they are qualified, by an appropriate combination of education and experience, for any project on which they work or propose to work”, and on point 1.07 “Ensure realistic estimates of cost, scheduling, personnel, and outcome on any project on which they work or propose to work and provide a risk assessment of these estimates.” [15].

2.1 Skills Management

An evaluation of the author's current skills is provided in this section, including strengths and weaknesses that must be improved to carry out the dissertation.

The technical foundation is strong in technologies like Java, GitHub, and Sonargraph, and supported through academic, professional, and personal experience. However, other technologies and frameworks fundamental to this project, including Spring Modulith, Grafana K6, and Kepler, are at a start-up stage of understanding, having been mainly limited to reading tutorials or guides at the beginning of the project.

On the analytical side, there is a solid academic basis in evaluating software quality measurements, but this will need to be honed and utilised more practically in the experiments. Written communication skills also need strengthening.

To cover these gaps, a set of practical strategies has been developed. These include building experiential knowledge with Spring Modulith during the actual code migration, use of tutorials and documentation to gain knowledge regarding Grafana K6 and Kepler, and regular usage of these tools in the project itself. There is a regular writing and feedback cycle to improve communication.

In short, the primary skills to focus on improving are Spring Modulith, performance and energy measurement tools (Grafana K6 and Kepler), and written communication. Improving these areas is key to guaranteeing the project's success. Technically, becoming proficient with the application of these tools and frameworks will directly influence the experiments' overall quality and analysis. And in terms of delivery, concise presentation and firm assertion will enable the project findings to be easily comprehensible and well-liked by both technical and non-technical readers.

Setting planning of skill development alongside project implementation, the author is setting himself up to meet both the technical and communication requirements of the dissertation, making it possible to create work that is high quality, perceptive, and professionally valuable.

2.2 Project Planning

This chapter of the dissertation outlines how the project is organised and managed to ensure its successful delivery. It defines the key stakeholders, costs, assumptions and constraints, risks, and a detailed project plan with activities and schedules.

The main stakeholders are the researcher and the supervisor; both are highly involved. The researcher is responsible for executing the work, while the supervisor ensures it meets academic standards. Other stakeholders who are relevant but less directly involved are open-source communities, software engineers, architects, professionals in industry, and all of them are interested in the result for practical application and adoption of modularity in monolithic systems.

The project does not have any significant direct costs since it is an academic project. However, there may be minor expenses (estimated between €50–€100) related to software licenses or accessing specific literature. Most tools and resources are expected to be available through institutional access or free online.

There are certain core assumptions supporting the project's feasibility: the selected open-source application is migratable and comparable; software like Spring Modulith, Sonargraph, Grafana K6, and Kepler will function as expected; there will be sufficient experimental data; and the timeline will stay on track.

On the other hand, some constraints set limits: a bounded schedule, a restricted budget, dependence on the supervisor's feedback, availability of suitable open-source software, and compliance with university rules on formatting. These are realistic constraints, but had to be kept in mind during planning.

The Work Breakdown Structure (WBS) of this dissertation is available in Appendix A. Those parts are:

- Dissertation Preparation (Sep 2024 – Jan 2025): Includes background research, writing research questions, and planning deliverables.
- Experiment Preparation (Feb – Mar 2025): Involves selecting an open-source app, defining a migration strategy, setting up the environment, and improving tool proficiency.
- Migration Phase (Mar – Apr 2025): Focuses on applying Spring Modulith to modularise the app, validating functionality post-refactor.
- Evaluation Phase (Apr – Jun 2025): Includes analysing maintainability, performance, and energy efficiency, and drafting the report.
- Conclusion Phase (Jun – Jul 2025): Final report writing, validation, presentation preparation, and delivery.

The timeline is tracked using Microsoft Project, with milestones clearly defined and monitored using progress indicators (% Complete), helping to identify and respond to delays early.

3 Background

This chapter introduces several essential concepts that, although not directly related to the objective of this work, provide crucial context necessary for fully understanding it. Section 3.1 presents the concepts of modularity, performance and energy efficiency in the context of software quality. Section 3.2 delves into what a monolithic application is and what a modular monolithic application is, clarifying what distinguishes traditional monolithic designs from modular monolithic architecture. Section 3.3 then explores the microservices architectural style. Section 3.4 gives insights about code smells, highlighting their implications in software applications. Finally, section 3.5, introduces event-driven design, explaining this architectural approach and its significance in software system design.

3.1 Software Quality

This topic addresses quality in software by exploring the software quality attributes Modularity, a sub-characteristic of Maintainability, as well as Performance and Energy Efficiency, which are both sub-characteristics of Performance Efficiency. Additionally, the tools Sonargraph (for modularity/maintainability), K6 and JMeter (for performance), and Kepler (for energy efficiency) are briefly introduced and explained.

Sonargraph is the only tool referenced for modularity/maintainability because it is the only one the author has had contact with. Adding, it is easy to use, available for the author in an academic context and the number of metrics that this tool offers.

The tools referenced in the performance section are only two tools; however, these tools were chosen to be compared and referenced because they are two of the recommended tools by the author's academic experience to perform load tests and because they are both free of charge.

Kepler is the only tool referenced for energy efficiency since it is the first time the author has contact with energy efficiency metrics. For this reason, and since the learning of the tool and its full capabilities are not relevant for this study, the author used the recommended academic reference tool to explore, learn and use it in the controlled experiment and because it is open-source and free to use.

3.1.1 Modularity

The quality of a software application can be represented by quality attributes. The standard ISO/IEC 25010 [8] is a product quality model that lists nine quality characteristics [9].

Each quality characteristic is composed of sub-characteristics. Modularity is a sub-characteristic of Maintainability [9], "[…] the degree of effectiveness and efficiency with which a product or

system can be modified to improve it, correct it or adapt it to changes in the environment, and in requirements".

The sub-characteristic modularity is the "degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components." [8].

In the 1970's Parnas mentioned the term "modularisation" in the computer science field, defining a "module" as a "responsibility assignment rather than a subprogram" [17]. It is also explained that the modularisation of a module is determined by specific design decisions (encapsulation), made at the outset for each independent module [17].

Modularity is directly related to the technical debt of a project [18]. Technical debt is the term that refers to the technical decisions made that hurt the maintenance of the project in the future, making it harder to add or update features [18].

In software systems, modularisation improves cognitive processing by organising code into meaningful units, such as classes or components. These modular structures are beneficial only when each unit is coherent and well-defined. If the groupings are disorganised or arbitrary, they lack cognitive coherence, which can hinder memory retention and understanding of the system. Therefore, a well-structured modular system can reduce technical debt and unnecessary complexity, making comprehension and memory alignment more intuitive [18].

To assess if the solution software architecture represents coherent and meaningful elements must be done qualitatively, supported by a group of measures and examinations: Cohesion through coupling - effective modularity in software systems depends on high cohesion and low coupling, where related subunits are grouped within modules to minimise external dependencies, cohesion is essential for modules to function well, while coupling should remain low to preserve modular integrity; names - naming conventions further aid modularity by clarifying each unit's single responsibility, and vague names may indicate unclear responsibilities; Well-balanced proportions - modular units should be proportionally balanced; overly large units may require decomposition to maintain system coherence. Imbalanced systems, where one unit contains most of the code, indicate poor modularity and can lead to inefficiencies [18].

Lilienthal, C (2022) presents a Modularity Mature Index (MMI), intending to help improve the solution's architecture and decrease its technical debt. This index has three main categories, with different percentage weights: Modularity - 45%, Hierarchy - 30% and Pattern consistency - 25%. Modularity has the most importance (hence the name), making it the most critical category because the other two categories, Hierarchy and Pattern consistency, are directly related to modularity [18].

Each category is further divided into subcategories, and each subcategory includes criteria that are evaluated through various methods. These methods may include architecture analysis tools, metric tools, and manual reviews, as some criteria cannot be quantified and must be assessed by reviewers to align with architectural goals [18].

3.1.1.1 Sonargraph

Sonargraph is a tool for static code analysis and software architecture evaluation, used by software developers and architects to enhance and maintain system quality. Its primary focus is on detecting and addressing architectural issues, commonly referred to as technical debt, through a group of features that include dependency analysis, code metrics, and architecture conformance checks [19].

The tool supports multiple programming languages, such as Java, C#, and C++ [20]. It allows for identifying architectural anti-patterns and code smells that can affect the system's maintainability and performance [21]. One of the greatest strengths of Sonargraph is its capability to impose architectural rules so that software systems conform to the intended design principles. This is especially useful in identifying and fixing architectural problems that arise as systems expand and start to diverge from their original design [22].

Another aspect that Sonargraph accommodates is dependency management, in which it possesses powerful visualisation and efficient software dependency management. Minimising unnecessary coupling and encouraging modularisation, it enables the creation of maintainable and scalable systems [23]. Additionally, Sonargraph provides detailed quality metrics and generates comprehensive reports, including technical debt indexes, which allow teams to prioritise refactoring tasks and systematically improve software quality [20]. These insights help identify areas of code complexity and dependency violations, allowing for guided refactoring and optimisation [24].

Compared to similar tools such as SonarQube and Structure101, Sonargraph distinguishes itself through its emphasis on architectural conformance and dependency visualisation. While SonarQube is widely recognised for its strength in code quality metrics, Sonargraph’s detailed focus on architecture-centric analysis makes it a preferred choice for confronting complex architectural challenges [25].

3.1.2 Performance

ISO/IEC 25010 contains “Performance Efficiency” as a quality attribute. It includes three sub-characteristics, which are time behaviour, resource utilisation and capacity [8].

According to ISO/IEC 25010, performance efficiency is defined as:

> “The degree to which a product performs its functions within specified time and throughput parameters and is efficient in the use of resources (such as CPU, memory, storage, network devices, energy, materials...) under specified conditions” [8].

The three sub-characteristics in Performance Efficiency by ISO/IEC 205010 [8] are categorised as:

- Time Behaviour: Focuses on the response time and processing speed of a system under specific workloads. It ensures that users experience minimal delays, making it essential for real-time systems and applications with stringent latency requirements.

- Resource Utilisation: Examines how efficiently the system uses computational resources such as CPU, memory, and storage. This directly impacts energy consumption and system sustainability.

- Capacity: Measures the system's ability to handle growing demands, such as concurrent users or transaction volumes, without significant performance degradation.

The taxonomy of performance by Barbacci et. al. (1995) elaborates on such specifics in terms of key problems and methods [26]. Latency would not be simply about average response times but rather more significant concepts like response windows and deadlines specifying the tolerance of the time window to handle events, and throughput is the difference in processing rates. In addition, it is how the system behaves under different loads since it should prioritise the most critical processes over the least critical.

3.1.2.1 Grafana k6

Grafana k6 is an open-source load-testing tool designed to evaluate and ensure the reliability and performance of modern software systems. This tool uses scripting so that the developers are allowed to write test cases in JavaScript, making it intuitive for those familiar with web development technologies. The primary purpose of K6 is to provide information on how systems perform under various conditions, enabling users to identify bottlenecks and areas of inefficiency [27], [28].

Key features of Grafana K6 include [27]:

- Simplicity and Developer Focus: The JavaScript-based scripting makes it easy to create, customise, and extend test scenarios.

- Scalability: K6 can simulate thousands of concurrent users, providing a realistic test environment for modern web applications and APIs.

- Integration with Grafana: Performance metrics gathered by K6 can be integrated into Grafana dashboards, offering robust visualisations for analysis.

- Extensibility: With support for plugins and libraries, K6 can be configured to meet diverse performance testing needs.

The tool is ideal for performance testing APIs, microservices, and other system components, ensuring they meet scalability and reliability expectations [28].

3.1.2.2 Apache JMeter

Apache JMeter is an open-source tool designed to test the performance and functional behaviour of web applications and other services. It is built in Java and provides an intuitive

graphical interface for creating, configuring, and executing performance tests. JMeter is highly extensible, offering plugins for advanced testing scenarios [29].

Core capabilities of Apache JMeter include [29]:

- Load Testing: JMeter can simulate heavy loads on a server, network, or object to test its performance under different conditions.
- Protocol Support: It supports a wide range of protocols, including HTTP, HTTPS, FTP, JDBC, and SOAP, making it versatile for testing various systems.
- Real-Time Reporting: JMeter provides robust reporting and visualisation tools to analyse test results.
- Customizability: Users can create complex test plans with support for scripting in Groovy and BeanShell.

JMeter is particularly popular for testing the load and scalability of web applications, ensuring they perform optimally under peak demand [29].

3.1.3 Energy Efficiency

Energy efficiency, as emphasised in the ISO/IEC 25010 standard, is closely linked to the broader concept of performance efficiency. It focuses on the system's ability to perform its functions while reducing energy consumption. Energy efficiency is particularly significant in the context of the resource utilisation sub-characteristic, which highlights the use of resources like CPU, memory, and energy under specified conditions [8].

The importance of energy efficiency has grown with the presence of mobile computing platforms (such as smartphones and tablets) and the rising operational costs of data centres. These systems are often energy-constrained, where energy-inefficient software can drain device batteries fast or contribute to high energy costs in large-scale infrastructures [9], [11].

Energy efficiency is required for both mobile computing and cloud infrastructures. Mobile systems are generally limited in energy, and inefficiency leads to battery deterioration. Energy waste at the scale of big infrastructures, such as data centres, can be problematic and environmentally unsustainable. It is believed that this energy waste accounts for as much as 10% of global consumption [11].

For instance, computation-efficient algorithms can reduce the number of operations required, reducing energy usage [9]. Resource monitoring, allocation, and adaptation are some of the specific techniques which play a critical role, allowing efficient energy utilisation. Resource monitoring through techniques such as metering or classification allows real-time monitoring of energy, whereas resource allocation mechanisms minimise waste in the consumption of resources [11].

Energy efficiency is no longer an unimportant topic but a fundamental quality attribute that impacts both the user experience and environmental sustainability. In applications which are used in mobile devices, low energy efficiency, meaning it consumes more energy than needed, can lead to bad reviews from users and reduce their choice and usage of them, while in data centres, it directly influences operational costs and carbon emissions [9]. Software architects must prioritise energy efficiency by integrating specific design concepts and energy tactics to navigate trade-offs effectively, ensuring that systems meet the dual goals of functionality and sustainability [11].

3.1.3.1 Kepler

Kepler, short for Kubernetes-based Efficient Power Level Exporter [30], is an open-source tool developed under the Cloud Native Computing Foundation (CNCF) to measure and report power consumption metrics in cloud-native environments. It is designed to address the growing need for understanding and optimising energy usage in Kubernetes clusters.

Kepler operates by monitoring container workloads and linking them to energy consumption at the node level. This data is derived from infrastructure telemetry, including CPU, memory, and network usage, and is used to calculate approximate power consumption [31]. Kepler features include:

- Granular Energy Metrics: Kepler provides detailed insights into the energy usage of containerised applications, offering visibility at both the node and container levels.
- Observability Integration: Seamless integration with tools like Prometheus and Grafana allows users to visualise energy consumption metrics alongside other performance data.
- Machine Learning-Powered Estimation: To enhance its functionality, Kepler employs pre-trained machine learning models to estimate energy consumption when direct hardware power metrics are unavailable.
- Cloud-Native Design: As a lightweight solution built specifically for Kubernetes, Kepler is scalable and introduces minimal overhead to monitored systems.

Kepler helps organisations to make informed decisions about workload placement, resource optimisation, and infrastructure scaling, since it enables real-time monitoring of energy consumption, contributing to operational efficiency and sustainability [31]. This aligns with the broader goals of ISO/IEC 25010, particularly the resource utilisation sub-characteristic of performance efficiency.

3.2 Monolithic Application

A monolithic application is an application that contains just one codebase/repository, aggregating multiple services to external systems or consumers using different interfaces. This codebase is shared by multiple developers, which must certify that when something is changed,

all the services do not stop working [32], [33], [34]. This codebase can be deployed on a single-server environment or in a multi-server environment, in case of the use of a load-balancer [32].

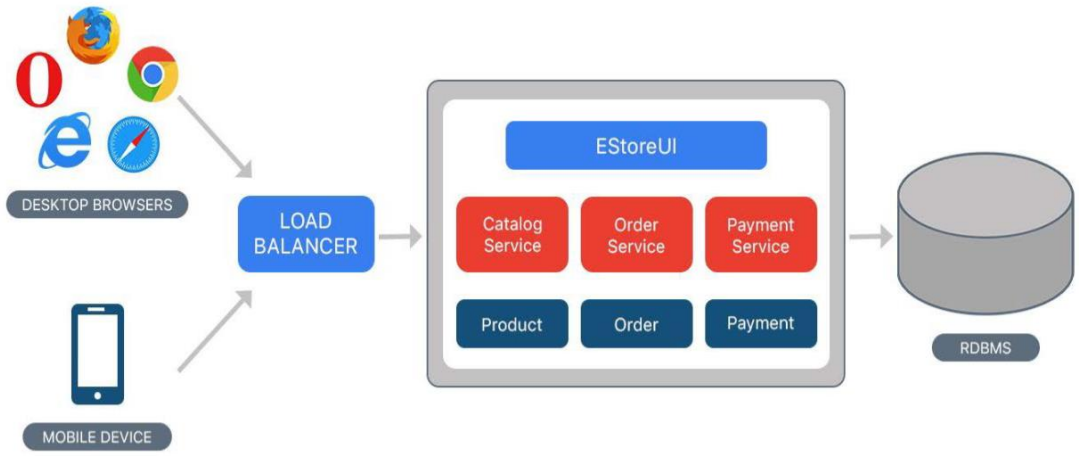


Figure 1 – Example of a Monolithic application
Source: [35]

The Figure 1 illustrates an example of an e-commerce monolithic application, providing a visual representation of the all-in-one codebase. This figure helps to conceptualise how monolithic applications are typically structured, making their design more accessible and understandable.

In a monolithic application, "different components (such as authorisation, business logic, notification, etc.) [are] combined into a single program from a single platform". This means that all the proposed components must be in the same technology, lowering flexibility [35].

3.2.1 Modular Monolithic

Su and Li (2024) made a systematic grey literature review with the Research Question "*What is modular monolithic*". By their definition, modular monolithic architecture combines the simplicity of monolithic structures with the modular advantages of microservices, which are introduced after. This pattern organises systems into independent, loosely coupled modules with clear boundaries and defined dependencies. Each module is isolated, enabling independent development within a unified deployment. This architecture aims to find the middle ground between the non-modular monolithic and microservices. It emphasises modular reusability and business-focused organisation over technical layers, enhancing maintainability. A modular monolithic approach offers a flexible approach to the application design, allowing incremental migration to microservices if desired [7].

Su and Li (2024) also found six main characteristics of a modular monolithic [7]:

- Module Segregation – Each module is self-contained and includes its own layers, making it easier to develop and test independently.
- Loose Coupling and High Cohesion – Modules are loosely coupled and tightly organised. Communications happen with APIs, preferably using asynchronous methods for flexibility.
- Single Database Schema – The entire system has one database for all modules.
- Maintainability and Scalability – Easier to maintain the system complexity and future growth.
- Monolithic Deployment – The application is deployed on one virtual machine (VM), with all modules.
- Unified Application Process – The application runs as a single process, without strict data sharing boundaries.

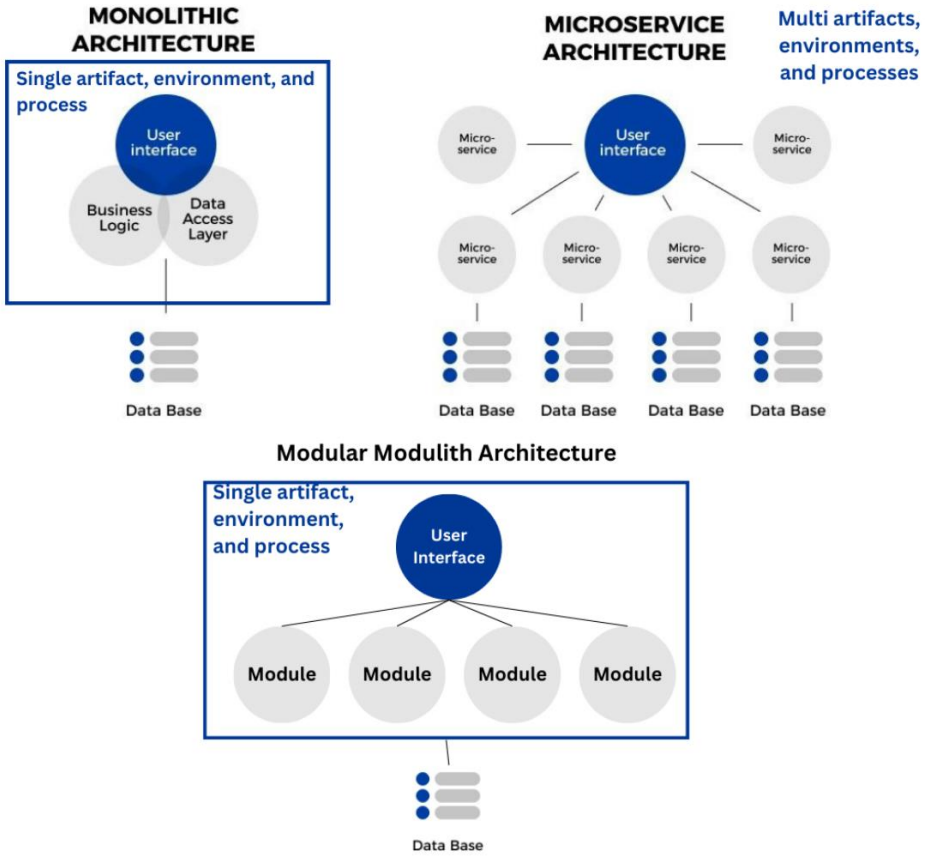


Figure 2 – Monolithic vs Microservice vs Modular Monolithic
Source: [36]

Figure 2 shows the difference between Monolithic Architecture, Microservice Architecture, and Modular Monolithic Architecture, helping to visualise how a modular monolithic differs from a non-modular monolithic and a microservice.

In contrast to non-modular monolithic systems, each module should have its own set of unit tests to ensure that its functionality works as intended independently. Integration tests should also be considered to test the interaction between modules [7].

There is one project and two frameworks referenced in [7] that allows developers to build modular monolithic, those being Spring Modulith, Service Weaver and Light-hybrid-4j.

These three frameworks/projects have been chosen to be compared since they are referenced in [7]. For this reason, they are explained in this document to understand how frameworks can help to leverage modularity in applications.

3.2.1.1 Spring Modulith

Spring Modulith is a project developed by Spring, to use with Spring Boot applications that is designed for modular monolithic applications.

It provides tools and APIs for defining and validating logical modules within a Spring Boot application. Spring Modulith enforces a modular structure for Spring Beans, giving control over what each module exposes through public APIs. Modules are organised as functional units, often as sub-packages and internal implementations are encapsulated within sub-packages to manage type visibility across modules [7].

The first release supports advanced package arrangements, flexible selection of a set of application modules to be included in integration tests, transaction event publication, documentation, observability and passage of time events implementation [37].

3.2.1.2 Service Weaver

Service Weaver is an innovative open-source framework developed by Google for building and managing distributed applications, currently only available in Go [7]. It is designed to simplify the complexities associated with developing and deploying software that operates across multiple distributed systems. Through the coupling of modular software design's benefits with the ability to deploy without any difficulty, Service Weaver presents developers with a solid and effective way to optimise and scale their applications.

At its core, Service Weaver allows developers to write programs as a monolithic program made up of modules, which is compiled and published as a set of microservices. This design pattern solution addresses one of the most significant challenges in distributed systems—balancing codebase simplicity with running scalability. Developers can define components in their applications and specify how these components should interact. At runtime, Service Weaver automatically manages the placement and communication between these components, whether they run in the same process or across distributed systems [38].

The framework emphasises performance and efficiency by employing advanced compilation techniques. These techniques allow Service Weaver to transform the modular application code into optimised deployment units while maintaining high levels of abstraction in the development phase. This enables developers to focus on the logic and functionality of their applications without worrying about the complexities of inter-service communication or scaling [39].

Service Weaver also supports integration with cloud-native ecosystems, making it compatible with Kubernetes and other orchestration tools. This flexibility allows applications built with this framework can be deployed in different environments. The framework's design allows easy debugging and monitoring, offering tools that provide information about the system performance and operational bottlenecks [40].

However, on the 6th of June 2025, service weaver is going to be permanently frozen and the GitHub repository archived, which means no more new commits and releases will be made [41].

3.2.1.3 Light-hybrid-4j

Light-Hybrid-4j is a lightweight, high-performance framework designed to bridge the gap between monolithic architectures and microservices. It offers a hybrid approach, allowing organisations to modernise incrementally by integrating legacy systems with modern distributed service designs, helping organisations migrate to cloud-native applications and still taking advantage of their existing infrastructure [42].

The framework's design revolves around optimising performance and scalability. Light-Hybrid-4j achieves this by employing an event-driven architecture that promotes asynchronous interactions between components. This decouples services, with the result that services remain responsive and highly scalable in load-sensitive environments. Its light runtime further ensures small resource footprints, a critical factor for organisations where resource-efficient deployment is a major concern [7].

The framework is also a viable option for containerised environments. It integrates seamlessly with modern deployment pipelines, including Docker and Kubernetes, supporting businesses in adopting cloud-native practices. Developers can use pre-defined templates to set up projects quickly, making Light-Hybrid-4j accessible even for teams new to distributed systems [42].

One of the most important factors of Light-Hybrid-4j is its hybrid architecture, since it allows easy communication between legacy modules and microservices. Because of this, it allows businesses to develop and scale systems without having to change them all. Because of its modular architecture, components can be reused across services, increasing consistency and lowering development effort. Additionally, Light-Hybrid-4j features in-memory processing and lightweight protocols, which improve throughput, lower latency, and provide performance regardless of workloads.[43].

In conclusion, Light-Hybrid-4j allows developers to design scalable, efficient applications and at the same time maintaining compatibility with existing systems. Its lightweight runtime, event-

driven nature, and support for modern cloud-native tools make it a powerful framework for organisations to face the complexities of application modernisation.

3.3 Microservices

A microservice is a cohesive, standalone process that communicates through messaging [12]. A microservice architecture is a distributed system where every module functions as an individual service [12].

This architecture is therefore a strategy to develop a suite of different small services which interact with each other as a single application. Each module runs independently and communicates through lightweight mechanisms, such as an HTTP resource API [33].

Figure 3 – Example of a Microservice application
Source: [35]

The Figure 3 demonstrates the same example as the application of the Figure 1, but using a microservice architecture approach. This helps to visualise the difference between monolithic applications and microservices architecture.

The microservice architecture offers various advantages. By having independent modules running on different processes, the microservice architecture allows different usage of technology per module. If a module fails, the application keeps running, not affecting the whole application. The scalability tends to be easier also because it is possible to choose what module to scale, contrary to monolithic architecture, which needs to scale the application as a whole. The deployment is easier and independent, not affecting other services. Finally, it allows companies to align their architecture with their organisational structure [33].

3.4 Code Smells

Code smells are hints in code pointing to issues in the codebase. While not necessarily indicative of bugs or functional problems, code smells indicate suspect patterns that can lead to maintenance trouble, technical debt introduced, and lower-quality code. Fowler et al. (1999) originally introduced the term, describing smells as patterns that, while not incorrect, suggest potential weaknesses in design or implementation [44]. Examples include duplicate code, long methods, and overly complex conditionals, which can make code harder to understand and refactor.

Code Smells' characteristics:

- Subjective and Contextual: A construct might be considered a smell in one context but perfectly acceptable in another.
- Impact on Maintainability: The primary effect of code smells is on long-term maintenance, often making it harder to add new features or fix bugs.
- Non-functional Issue: Unlike bugs, code smells are related to the design and structure of code rather than its functionality.

According to Yamashita et al. (2012), code smell identification is essential for programmers who intend to ensure maintainable software systems. Identification would usually require expertise because tools might be employed as a first detection, but judgment is essential for ensuring their use in context appropriately [45].

3.4.1 Architectural Code Smells

Code smells appear in the program code, while architectural code smells focus on a software system's structure and design choices and target the whole system's design, negatively affecting scalability, extensibility, and resilience.

These smells are considered anti-patterns that appear because of poor architectural decisions and typically appear when the architecture does not meet the design principles, like modularity, low coupling or high cohesion, for example, cyclic dependencies, god components and unclear layering [46].

Architectural Smells' characteristics are the following:

- Broader Impact: Unlike code smells, architectural smells influence the system's structure and its ability to evolve.
- Difficult Detection: Identifying architectural smells often requires tools and expertise because they are more abstract than code smells.

- Higher Cost of Refactoring: Fixing architectural smells is usually more resource-intensive due to the systemic changes required.

Architectural smells, such as the violation of design principles, lead to challenges like increased complexity and reduced development speed. For instance, if components are not well-isolated, changes in one part of the system can have effects across the whole system, making refactoring and enhancements challenging [47].

3.5 Event-Driven Design

Event-driven design is a design strategy, often used in microservices, that implements inter-module asynchronous communication by a message bus using events [48], [49]. It is composed with the purpose of decoupling components [50].

An event in event-driven design is something that may be noticeable that happens in the business context, both inside and outside [51]. It can be any occurrence, activity or change in state, and is often processed with some form of delay, making it possible to ignore certain events, delay response or start processing in the moment [52].

In this design strategy, there is a producer who publishes an event or message into a queue of a distributed message broker, and then, there is a consumer which pulls the events that it is listening to and processes the logic needed [49]. Each module must implement its reactive logic to perform the tasks needed for specific events [48].

By this logic, this promotes loose coupling since the publisher only knows the event to publish and the consumer only knows the event to consume, removing the dependency between these two modules. However, the traceability for this kind of process can be difficult; for this reason, this works best with asynchronous flows of work [51].

The event-driven design can increase performance by decreasing response time, CPU and memory usage, and network consumption and increasing the number of responses of a system if the source code is well-optimised and structured, promoting the quality of the system overall [50].

The event-driven approach tends to the potential modules' scalability and resilient systems, allowing organisations to handle data at scale, providing flexibility in reacting to business requirement changes and improving overall communication efficiency [53].

Bellemare (2020) [53] explains that these modules should adhere to principles of domain-driven design, more specifically, bounded contexts, so that the responsibilities are cohesive and loose coupling is promoted.

Apache Kafka and RabbitMQ are the two tools which have been chosen to explain and compare since they are popular tools that help implement event-driven design. Both have big communities, good documentation, are free of charge and open-source.

3.5.1 Apache Kafka

Apache Kafka is an open-source distributed publish-subscribe messaging system developed by Apache that offers solutions to real-time problems dealing with real-time volumes of information, providing seamless integration between producers and consumers, by allowing producers to publish information without blocking it and without knowing who the consumers are [54].

Apache Kafka communicates via a high-performance TCP network protocol and can be deployed on hardware, virtual machines and containers in both on-premises and cloud environments [55]. The capabilities of Apache Kafka consist of [55]:

- To publish and subscribe to event streams, enabling continuous data import and export from other systems.
- To reliably and durably store the streams of events.
- And finally, to process streams of events when they occur or retrospectively.

Apache Kafka implements the bullet points mentioned before with the following characteristics [54]:

- Persistent messaging: Apache Kafka has an O(1) Complexity disk structures that provide constant-time performance.
- High throughput: It can handle millions of messages per second.
- Distributed: Supports message partitioning over their servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.
- Multiple client support: It allows multiple integrations with many different platforms such as Java, .NET, PHP, Ruby and Python.
- Real-time: Messages generated by producer threads should be instantly available to consumer threads.

Apache Kafka is, therefore, a robust and highly effective platform to implement event-driven architectures. It offers powerful tools for managing asynchronous communication and data flow [56].

3.5.2 RabbitMQ

RabbitMQ is an open-source message broker built in Erlang that acts as middleware for different applications, implementing Advanced Message Queueing Protocol (AMQP) [57], [58]. It allows smooth asynchronous intercommunication between modules based on messages.

These messages are loosely coupled, meaning that the sender and the receiver do not need to be running at the same time [58].

RabbitMQ can be used and deployed in various technologies such as C#, PHP, Ruby, and Python [57], [58] and the messages are communicated over TCP connections [58].

RabbitMQ has three key characteristics [59]:

- Interoperability: RabbitMQ supports several open standard protocols and can be used with several programming languages.
- Flexibility: RabbitMQ provides a diverse set of solutions to implement how messages should be communicated from the publisher to the consumer, such as routing, filtering, streaming, federation and more.
- Reliability: RabbitMQ can acknowledge message delivery and replication of messages across a cluster.

Roy (2018) states that this technology enables operational flexibility, which is difficult to achieve without the loose coupling that RabbitMQ offers [60].

4 Literature Review

This chapter summarises the state of the art about the subject in this document. Understanding how modularity affects monolithic architectures is the main goal, with a focus on software quality attributes like performance, maintainability, and energy efficiency. This chapter identifies the major contributions, gaps, and difficulties that guide the research questions and methodology of this dissertation through a methodical analysis of earlier studies.

To ensure clarity and depth, the chapter is structured into seven sections for clarity and depth. Section 4.1 begins by clearly defining the research questions and outlining the data sources utilised. Following this, section 4.2 describes the process of constructing the search terms used to gather relevant literature. Section 4.3 then provides a detailed explanation of the inclusion and exclusion criteria, ensuring the selection of high-quality and relevant studies. In section 4.4, the chapter elaborates on the system review process, describing the set of guidelines adopted. After, section 4.5 presents the findings obtained from this systematic review. These results are further synthesised in section 4.6, addressing how they relate to the research questions, and highlighting the implications of these insights. Finally, Section 4.7, offers a conclusion about these results, summarising the key insights obtained from the review.

4.1 Research Questions

Based on the problem and description already mentioned, the research questions developed for this study are the following:

RQ1: How does modularity impact a monolithic architecture on software maintainability, performance, and energy consumption?

RQ2: What primary technical factors influence an organisation's choice between microservices and monolithic architecture?

RQ3: How does transitioning from a non-modular to a modular monolithic architecture affect software maintainability, performance, and energy consumption?

To answer these research questions ACM Digital Library and IEEE Explore were used to find relevant and quality studies about the topic. The data sources listed offer many different types of publications, such as journal articles, books, conferences, and newspapers. These data sources have high-quality, peer-reviewed publications available that can be relevant in the information technology area, thus distinguishing themselves from the rest.

4.2 Search Terms

In this section, a search query is developed that helps to find the most relevant studies for the topic. The main keywords used are monolithic architecture, microservice architecture, quality attributes, maintainability, performance, energy consumption, architecture migration, Spring Modulith, and modularity.

As each research question has its purpose and distinct researchable content, three search queries were developed to obtain relevant studies.

For RQ1, the following search terms defined are:

(“monolithic architecture” OR “monolithic software” OR “monolithic system”) AND (“High modularity” OR “Modular architecture” OR “Modular system” OR “Modularization” OR “Modularity”) AND (“Impact” OR “Effect” OR “Change”) AND (“Maintainability” OR “Performance” OR “Energy consumption”).

These search terms include the key concepts for the first research question, including the three architectural styles, the modularisation of the system, followed by the verb of action, meaning their impact, effect, change, and then the three software quality attributes explained in this document.

For RQ2, the search terms defined are:

(“Microservices architecture” OR “Microservice architecture”) AND (“monolithic architecture” OR “monolithic software” OR “monolithic system”) AND (“Choice” OR “Decision”) AND (“architecture trade-offs” OR “architecture evaluation” OR “decision-making” OR “challenges” OR “comparison”) AND (“Maintainability” OR “Performance” OR “Energy consumption”).

The terms include the key concepts of the research question, searching for the decision reason to switch from the monolithic architecture to the microservices application and for the comparison between the two in terms of maintainability, performance and energy consumption.

Finally, for RQ3, the search terms are the following:

(“Transition” OR “Migration” OR “Transformation” OR “Change” OR “Adaptation” OR “Transform” OR “Update” OR “Modularization”) AND (“non-modular monolithic” OR “non-modular monolith” OR “traditional monolithic” OR “traditional monolith” OR “monolith systems” OR “monolithic application”) AND (“Modular monolithic” OR “modular monolith” OR “modules” OR “Modularity” OR “monolith modularization”) AND (“Maintainability” OR “Performance” OR “Energy consumption”).

This research question focuses on the effects of maintainability, performance and energy consumption when migrating from a non-modular monolithic to a modular monolithic.

The search terms were combined into one search query, which was used in the digital libraries to search for high-quality papers:

((“monolithic architecture” OR “monolithic software” OR “monolithic system”) AND (“High modularity” OR “Modular architecture” OR “Modular system” OR “Modularization” OR “Modularity”) AND (“Impact” OR “Effect” OR “Change”) AND (“Maintainability” OR “Performance” OR “Energy consumption”)) OR ((“Microservices architecture” OR “Microservice architecture”) AND (“monolithic architecture” OR “monolithic software” OR “monolithic system”) AND (“Choice” OR “Decision”) AND (“architecture trade-offs” OR “architecture evaluation” OR “decision-making” OR “challenges” OR “comparison”) AND (“Maintainability” OR “Performance” OR “Energy consumption”)) OR ((“Transition” OR “Migration” OR “Transformation” OR “Change” OR “Adaptation” OR “Transform” OR “Update” OR “Modularization”) AND (“non-modular monolithic” OR “non-modular monolith” OR “traditional monolithic” OR “traditional monolith” OR “monolith systems” OR “monolithic application”) AND (“Modular monolithic” OR “modular monolith” OR “modules” OR “Modularity” OR “monolith modularization”) AND (“Maintainability” OR “Performance” OR “Energy consumption”)).

Based on the Search Terms mentioned, combining the operators **OR** and **AND**, the search made on the data sources chosen, for each search query, only returns publications with the desired content related to the data queried.

The research methodology adopted was the *Trail-and-Error Search* method. This is a method that starts with a base search query and improves it based on the results, refining it after each iteration [61]. The search query mentioned before is the final version, after being revised and adapted.

4.3 Inclusion and Exclusion Criteria

The inclusion criteria decided for the articles selected are:

- **IC1** – Studies about the use and comparison of modular monolithic, non-modular monolithic and microservices.
- **IC2** – Studies about the impact of modularity on maintainability, performance and energy consumption.
- **IC3** – Studies about the effects of transitioning from a non-modular architecture to a modular monolithic architecture.

The exclusion criteria for the articles are:

- **EC1** – Studies which are not available in English.
- **EC2** – Studies without the full text available.
- **EC3** – Studies without an abstract.
- **EC4** – Studies before 2000.

- **EC5** – Studies that mainly focus on the DevOps area.
- **EC6** – Studies that do not focus on software development and architecture.
- **EC7** – Studies that only focus on microservices application development.

4.4 Guidelines for Literature Review

The set of guidelines used to complete this literature review is based on the PRISMA systematic review process [62]. PRISMA stands for Preferred Reporting Items for Systematic reviews and Meta-Analyses, and it follows three steps:

- Identification: Gather all the potential studies that could be relevant to your research questions, following the eligibility criteria
- Screening: Review the studies identified in the previous step to determine whether they meet the inclusion criteria. In the first phase, articles are categorised as “Relevant” or “Irrelevant”. In the second phase, the remaining articles are carefully analysed to see if they are relevant to the research questions; if not, they are discarded.
- Included: The final step includes the total of articles used to answer the research questions.

With these steps, using the search queries and the Inclusion and Exclusion criteria, it is possible to gather relevant information to answer them.

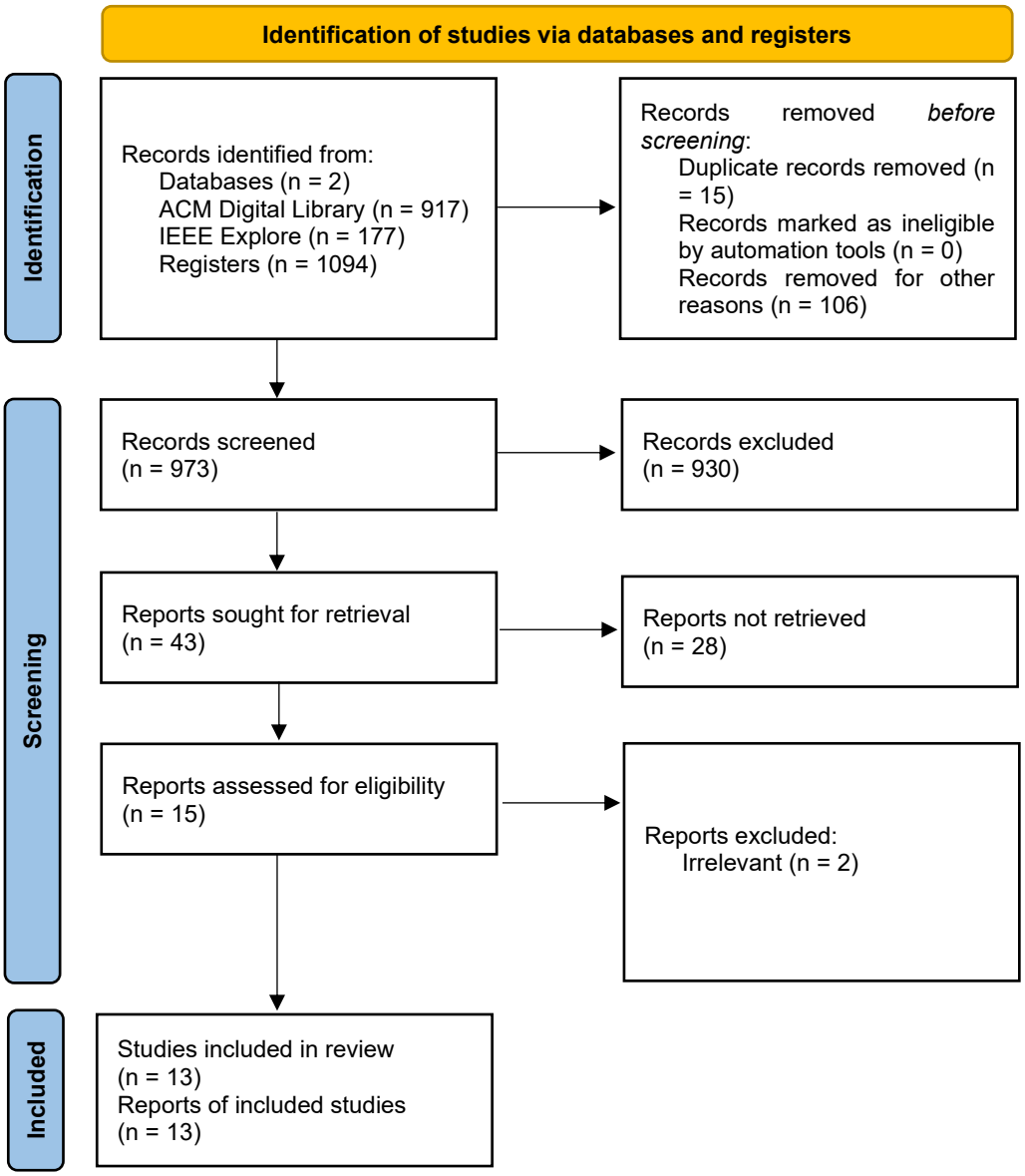


Figure 4 – PRISMA systematic

The process made by using the search query mentioned is represented in Figure 4.

4.5 Results

After the identification and screening were completed, the articles in Table 1 were included to gather information about the research questions.

Table 1 - Articles used to answer research questions

Research Question	Articles
RQ1	[7], [63], [64], [65], [66], [67]
RQ2	[68], [69], [70], [71], [72]
RQ3	[64], [65], [66], [73], [74]

4.6 Discussion

The architecture of software systems plays an important role in affecting their maintainability, performance, and energy efficiency. In recent years, the debate between adopting monolithic or microservices-based architectures has gained significant attention, driven by advancements in modular design principles and the growing need for scalable, sustainable systems. Monolithic architectures remain a relevant choice for most companies, even though microservices offer decentralised and distributed advantages.

Three basic topics of this architecture are explored in the following sections. First, it explores the impact of modularity on monolithic software architectures and its impact on maintainability, performance and energy consumption. Then, it explores the technical factors that influence the choice from companies' choice between microservices and monolithic designs. Finally, it is analysed what implications there are when companies want to migrate from a non-modular to a modular monolithic system.

4.6.1 RQ1: How does modularity impact a monolithic architecture on software maintainability, performance, and energy consumption?

Modularity of monolithic systems affects software maintainability, performance, and energy efficiency. These effects can differ because of the choice of modularisation, the contexts, and the compromises that must be accepted to achieve higher modularity.

In software engineering, modularity offers advantages in terms of maintainability. The modular-monolithic approach, which divides the application into well-defined modules, gives developers tools to manage dependencies, minimise the number of changes, and lower the chance of failure for modules that rely on one another, improving maintainability by addressing anti-patterns and dividing large, monolithic applications into smaller, more manageable pieces [64]. Modularisation supports better separation of concerns, simplifies debugging, and accelerates onboarding for new developers. However, refactoring the codebase to increase modularity can introduce abstractions, which, sometimes, can negatively affect energy efficiency [64]. Modular monolithic desgins can be seen as a middle step between non-modular monolithic designs and microservices by enabling incremental scalability [7]. The modularity of the monolithic architecture also helps reduce the costs and complexities associated with full microservices

adoption, particularly in cases where the benefits of a complete migration are not immediately clear [7].

Performance, however, has its pros and cons. It is possible to improve response times and memory consumption by using modularisation to get rid of architectural code smells. But it can be impacted by identifying bottlenecks and design flaws when architectural code smells are eliminated [66]. However, there is the possibility of over-modularisation, which could result in "software bloat". This phenomenon occurs when abstraction layers, added for modularity, introduce runtime overheads such as increased memory usage and slower execution [63]. Therefore, there should be consideration when balancing modularisation benefits and performance.

Energy consumption presents an additional layer of complexity in the modularisation discourse. While modular designs are generally associated with better maintainability and scalability, their impact on energy efficiency is mixed. Dhaka and Singh (2016) explore the relationship between code smell elimination and energy consumption, finding that while refactoring improves code quality, it can also increase energy costs [65], as it is possible to analyse in Table 2. In this study, the refactoring of the code smells god class (G), feature envy (F) and long method (L) in different orders and calculating architectural metrics such as LOC (Lines Of Code), NOC (Number Of Classes), NOM (Number Of Methods), CBO (Coupling Between Objects) and CC (Cyclomatic Complexity) which are correlated with maintainability and modularity, gave higher energy consumption as referenced before.

The suffix "_O" after the application acronym shows the original code, and the other suffixes are the source code of different applications with architectural code smells refactored.

Table 2 - Architectural Metrics and Energy Values

Source: [65]

Version	LOC	NOC	NOM	CBO	CC	Energy (mJ)
JH_O	73231	552	5008	513	1.33	64230
JH_FGL	76755	624	5520	590	1.33	112307
JH_FLG	77002	652	5540	595	1.33	115390
JH_GFL	76196	624	5434	562	1.33	84900
JH_GLF	76384	624	5463	564	1.33	85560
JH_LGF	76700	633	5513	574	1.33	85780
JH_LFG	76550	631	5488	569	1.33	85550
CB_O	71767	322	3118	299	1.99	163580
CB_FGL	73393	352	3270	328	1.95	170230
CB_FLG	73423	352	3275	328	1.95	171850
CB_GFL	72813	345	3229	319	1.96	165020
CB_GLF	73154	345	3263	319	1.95	167010
CB_LGF	73168	348	3264	324	1.95	169700
CB_LFG	72854	345	3236	321	1.96	169480
CI_O	55473	262	2648	256	1.77	82250
CI_FGL	56220	282	2751	272	1.75	83718
CI_FLG	56299	282	2755	272	1.74	83790
CI_GFL	56135	280	2729	270	1.75	83280
CI_GLF	56229	280	2740	270	1.75	83390
CI_LGF	56198	280	2746	270	1.75	83650
CI_LFG	56219	278	2736	268	1.75	83404

When adding more modules, some overhead is introduced by additional abstractions, which can inflate energy usage. Similarly, Pérez-Castillo and Piattini (2014) [64] note that the effort to resolve anti-patterns such as "god classes", which increase modularity and maintainability, may increase power consumption due to the processing demands of modularised components. Thus, the energy efficiency of a modular system depends on the specific design choices and their alignment with operational goals.

Gravanis et al. (2021) contribute one more perspective in this debate through examples where even if monolithic architectures are viewed as legacy, they can be suitable for today's software needs because the motivation to modularisation and to move to microservices in some cases comes from trends over actual business imperatives, adding undue complexity and consumption of resources. Their research outlines why modularity must be thoroughly evaluated when it is required, and its adoption must be driven by the necessity of operation rather than by industry trends [67].

In conclusion, modularity in monolithic design has advantages to maintainability. Modularity makes it easier for organisations to handle complexity and develop future-proof designs. On the other hand, performance and energy consumption trade-offs promote the need to do a business requirements analysis before implementing this architecture. Fontana et al. (2023) and

Gravanis et al. (2021) [66], [67] refer to the need for architectural smells to be addressed and modularity to be evaluated, while Pérez-Castillo and Piattini (2014) and Dhaka and Singh (2016) [64], [65] caution against overlooking the costs of modularisation, particularly with the usage of energy.

4.6.2 RQ2: What primary technical factors influence an organisation's choice between microservices and monolithic architecture?

Organisations are increasingly shifting from monolithic to microservices architectures due to a variety of technical and organisational benefits that address the limitations of monolithic systems.

The increasing complexity and maintenance issues that come with large monolithic applications are two of the primary factors behind the migration from monolithic architectures to microservices. As the business grows, maintaining a monolithic system becomes more challenging. Coupling everything in a single application makes changing and updating riskier and slower. In their case study, Hayretci and Aydemir (2021) highlight how legacy banking systems, which were originally monolithic, became increasingly hard to maintain as they grew. Their findings suggest that monolithic systems, while initially easier to develop, eventually become difficult to scale, especially when trying to adapt to new technological advancements [70]. In an empirical investigation [68], maintainability is the most desired improvement when migrating from monolithic towards microservices. In another study [71], organisations also favour microservices rather than monoliths because of their easier maintenance and evolution, and other quality attributes directed to maintainability, such as high cohesion and low coupling.

Another important factor driving migration is the limitations of monolithic architectures in terms of scalability. When an organisation's user base or traffic increases, scaling a monolithic system frequently necessitates replicating the entire application, which results in inefficiency and increased expenses [69], [70]. Additionally, there is a greater chance that modifying the code can affect the entire system, necessitating a redeployment of the whole system. [72]. Furthermore, if there is any component that is bottlenecking the application in a monolithic system, the whole system needs to be scaled [68]. Microservices, on the other hand, provide a more detailed approach to scaling. Each service can be scaled separately based on demand, allowing organisations to make the most of resources [68]. This benefit is brought out where shifting business requirements are likely to require independent scaling of specific services, for example, in the banking business, like payment processing, account management, and fraud protection [70]. Bucchiarone et al. (2018) report on the migration efforts in banking, where microservices offered flexibility in scaling individual components of the system without the need to scale the entire application [69].

The monolithic approach seems not to be the solution in certain areas like Internet of Things (IoT), given the need to build, deploy, implement and scale the system [72]. In other studies, [68], [71] most companies have in mind scalability, DevOps support and independent and automated deployment as some of the most valuable technical factors as well.

Another technical factor is the fault tolerance that microservices offer. In a monolithic architecture, a failure in one part of the application can potentially bring down the entire system. With microservices, on the other hand, the impact of a failure is typically isolated to the individual service, allowing the rest of the system to continue functioning [68]. This fault isolation and the ability to deploy services independently are especially important in high-availability environments [69]. Bucchiarone et al. (2018) point out that microservices provide enhanced fault tolerance, as failures in one service can be contained, preventing cascading failures across the entire system [69]. In IoT, the need for fault-tolerant architectures is a motivator to have this architecture in mind [72].

Organisations often migrate to microservices to address legacy modernisation challenges. As systems grow older, they may lack the flexibility required to integrate with modern technologies or adapt to new business processes [70]. Modernising legacy systems by refactoring them into microservices is seen as a solution to this problem. Hayretci and Aydemir (2021) discuss how legacy banking systems are refactored into microservices to enable integration with newer technologies, improving the agility and flexibility of the organisation [70].

Based on the studies in this literature review [68], [69], [70], [71], [72], the most valuable technical factors when choosing microservices over monolithic are maintainability, scalability and fault tolerance.

4.6.3 RQ3: How does transitioning from a non-modular to a modular monolithic architecture affect software maintainability, performance, and energy consumption?

The transition from a non-modular to a modular monolithic architecture represents an important change in software development practices, with big implications for maintainability, performance, and energy consumption.

Software systems that are modularised are easier to maintain. Tightly coupled components can hurt non-modular systems, making it challenging to comprehend written code, debug it, and add new features as the system expands. When modularising a large-scale banking application, modularisation provides a clear separation of concerns by organising the system into independently manageable modules, each with well-defined boundaries and interfaces. This approach reduces inter-module dependencies, facilitates debugging, and accelerates onboarding for new developers by localising the learning effort to specific modules [74]. Similarly, Faustino et al. (2024) cite that transitioning to a modular monolithic as an intermediate architectural step alleviates many of the structural issues connected to monolithic systems; for this reason, improving maintainability even before considering full-scale migration to microservices [73].

The impact of modularisation on performance depends on the context given. Modular architectures promote optimised design practices by allowing developers to focus on high-cohesion modules. Fixing architectural smells, such as cyclic dependencies and dependencies,

by applying modularisation can improve response time and decrease memory usage. Refactoring modular components can decrease method execution times by up to 47% and reduce memory consumption by 20%, showing the performance potential of modularised systems in the right way. However, the introduction of module interfaces in modular monolithic applications adds a layer of communication overhead, which may initially decrease performance compared to tightly integrated non-modular systems [66]. Faustino et al. (2024) acknowledge this trade-off but argue that the long-term benefits of modularisation, such as improved maintainability and better resource allocation, win over these initial performance costs [73].

Energy consumption emerges as a critical consideration in the context of modern software systems, particularly as sustainability becomes a priority in informatics technology. Dhaka and Singh (2016) explore the relationship between modularisation, code smells, and energy efficiency, noting that while refactoring and modularisation generally improve maintainability, they can have mixed effects on energy consumption. Their empirical work indicates that some refactoring techniques reduce energy consumption by deleting inefficient paths, but others, particularly those which increase inter-module communication, may increase energy consumption [65]. This is consistent with the work of Pérez-Castillo and Piattini (2014), who discuss the energy impact of refactoring god classes in monolithic systems. Their research identifies that while modularisation reduces architectural complexity, increased message traffic among modularised parts can level off some energy benefits of effective design methods [64]. All these discoveries focus on the problem of strategic planning within modularisation efforts, balancing energy-saving goals with architectural improvement.

In general, the transition from a non-modular to a modular monolithic architecture improves maintainability, addressing most of the problems associated with traditional monoliths. Performance consequences, although initially neutral for the benefit of the cost of communication, tend to level out as the return on modularised design becomes realised over the long term. Energy consumption has a multi-dimensional trade-off, with care to ensure optimisation so as not to make system inefficiency increase due to modularisation. These aspects make the modular monolithic a good design strategy, particularly in cases where a monolithic system is needed without compromising the flexibility of modular components.

4.7 Summary

The literature review explored the connections between software architectural designs, focusing on modularity in monolithic applications and microservices. The study introduced some of the impacts of modularity on core software quality attributes—maintainability, performance, and energy consumption—within the broader context of modern software engineering. The various choices organisations need to make in architectural migration are highlighted by an exploration of the trade-offs in monolithic versus microservices architecture, and intermediate practices of non-modular to modular monolithic architecture.

In monolithic applications, modularity became an important factor to improve maintainability. Dividing functionalities into distinct modules not only facilitates debugging but also ensures a clear separation of responsibilities by lowering the possibility of changes between dependent modules. Modularity may, however, come at the expense of energy usage and performance optimisation issues, requiring a trade-off between the advantages of abstraction and system effectiveness.

Although there is a short-term increase in communication overhead, the shift to modular architectures has been perceived to reduce architectural inefficiencies and performance bottlenecks. In terms of energy efficiency, modularisation can result in higher energy consumption because of more inter-module communications, even though it eliminates architectural code smells.

The positive points of microservices over the monolithic architecture are better fault tolerance, scalability, and freedom to use different technologies. The modular monolithic architecture offers a middle ground between the simple design of monolithic systems and the modularity and flexibility needed for the growing number of new business requirements.

The context when making decisions about the design is important and is analysed in this discussion. Monolithic modular applications cannot be the solution to every use case, but can offer advantages which justify their implementation for organisational and technical goals. Lastly, architectural migrations must be done through an assessment of priorities, trade-offs, and system needs over the long term. The balanced strategy ensures that architectural decisions meet immediate needs without constraining future scalability and innovation.

5 Experiment

This chapter discusses the selection and analysis of an application chosen to evaluate the impact of modularisation on performance, maintainability and energy efficiency. Additionally, it covers the analysis and architectural design of the selected project, including insights into its business context. The chapter is structured into three sections to enhance clarity and comprehensibility. Section 5.1 details the criteria used in selecting the project and briefly introduces the project chosen. Section 5.2 describes the necessary modifications made to ensure consistency and comparability among the different architectural styles evaluated. Finally, Section 5.3 concludes the chapter by summarising the main points discussed.

5.1 Project selection

In this section, the project selection process is explained. First, it presents the criteria for choosing the project, followed by an explanation of the project selected, including the business context and the architecture of the non-modular monolithic, modular monolithic, and microservices, accompanied by artefacts.

5.1.1 Criteria

The criteria for choosing the application are determined in this subsection. This is a crucial step in this work since it is the main foundation for performing the controlled experience. By choosing an application with the right characteristics, the results obtained and analysed have higher quality, leading to better conclusions.

At least one application must be present to evaluate and draw conclusions about the needed objectives to address the problem described and conduct the controlled experiment.

This project is based on an academic context, and it is not sponsored; therefore, there is a low (close to none) budget to develop this work. For this reason, the application chosen must be open source and should also be publicly available on a Git repository service, to be possible to share the results obtained.

Another important criterion is the license of the application. The license must allow the use, modification and distribution of the application to achieve full transparency. Any of the licenses that do not allow these actions may affect the results gathered, since there is a chance that the applications must be altered to focus on the study in context.

The application must follow the Object-Oriented Programming paradigm and domain-driven design. This criterion allows not only for the ease of use of the application, but since there is a comparison between a monolithic, modular monolithic and microservices, there should be the

possibility to separate the application into modules, creating boundaries, therefore the functionalities and domain concepts are clear.

Another important point to select the right application is that the application works, and has tests, both unit and integration tests. This ensures the application does not give wrong results, affecting the outcome of the experiment.

To achieve consistent and truthful results, there should be a minimum of complexity needed. The application must have at least three modules, being able to generate results that replicate somewhat near the business environment scale.

However, since the time to develop this project is limited, the application must not be too complex to understand and, if needed, to migrate. This work is not focused on the migration effort, but rather on the comparison of metric values between three different architectures: non-modular monolithic, modular monolithic and microservices. For this reason, the application chosen must at least be somewhat easy and simple to comprehend to lower the effort of migration and complete this task as fast as possible.

There is no constraint on the technology; however, Java with Java Spring Boot or C# with .NET is preferable, since the author has more knowledge compared to other technologies.

As referenced in this chapter, migrating the application is not the focus of this work. For this reason, an application which is already available in all three architectures being compared is highly preferable. By skipping the migration step, the time and effort needed to develop this task can be allocated to more important tasks, achieving higher quality in the most important areas for the scope of the project.

If it is not possible to choose an application which is already migrated, the application must be in a non-modular monolithic architecture, allowing the migration from non-modular monolithic to modular monolithic and to microservices, following the direction of application development and its lifecycle.

5.1.2 Project Chosen

Based on the criteria defined in the last section, the project chosen is Spring PetClinic, fitting all the criteria mentioned above.

Spring PetClinic is an open-source application developed by Spring and maintained by the community to showcase how the Spring framework works. It is publicly available on GitHub [75]. It is a community-driven project, meaning it is actively maintained and updated by the Java Spring community. There are a vast number of forks in this project that implement different architectures, including the non-modular monolithic, the modular monolithic, using Spring Modulith, and microservices.

This application, as it is community-driven, is up to date, with unit and integration tests, an important criterion for the project selection. The tests already developed for the application

show that the application has been properly tested. Additionally, manual testing was also performed by using the application and testing each one of the use cases, which will be shown further, defined in Figure 6.

All the artefacts and code adjustments from this section are publicly available on GitHub, in this repository [76].

5.1.2.1 Business Context

The Spring PetClinic application is an application that performs the management of small veterinary clinics.

This application is for a veterinary clinic that has recently started its operations and currently serves a limited number of clients. The clinic anticipates steady growth over the next two years and wishes to evaluate different architectural options proactively.

It offers important features to clinics intending to manage employees and appointments efficiently. Some of these are pet owner and pet tracking and having an easily maintainable record of their medical history, appointment scheduler, keeping the vets accessible for tracking visits, management of the medical records, storing the pet health information, improving the continuity of care and employee management by also storing the information of the vets, including information about their specialisation.

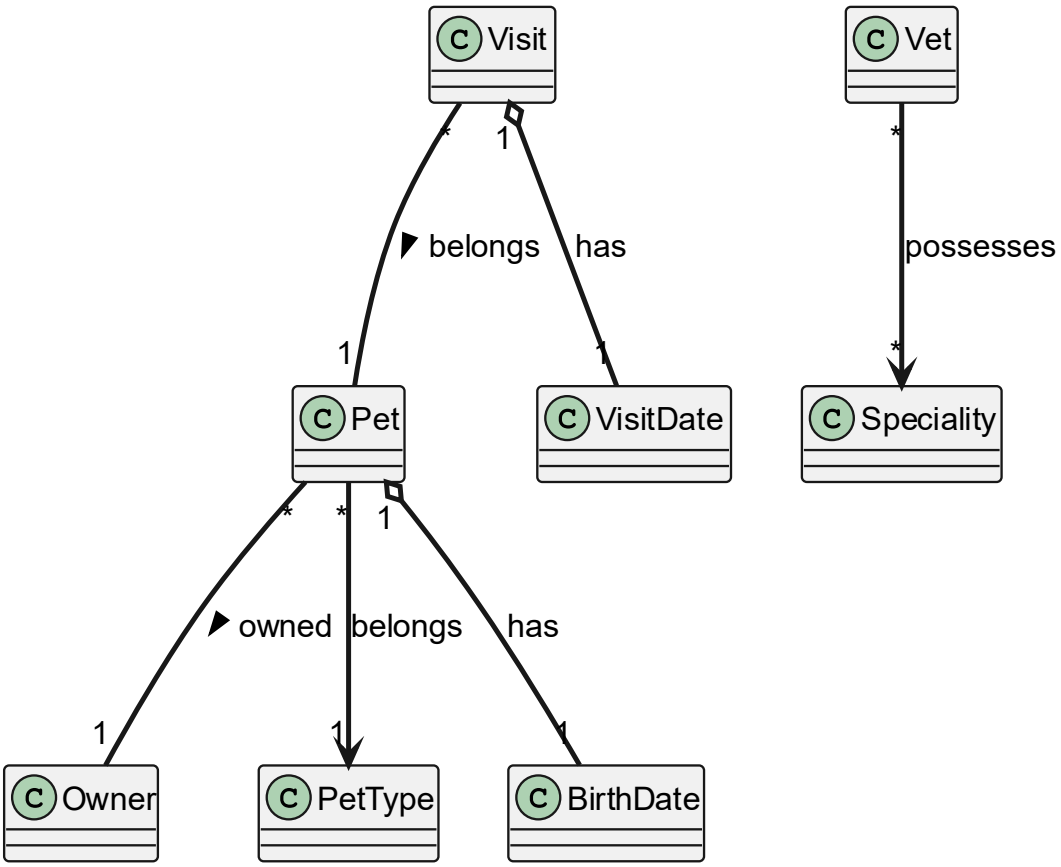


Figure 5 – Spring PetClinic Domain Model

Adapted From: [77]

In Figure 5, the business concepts of the Spring PetClinic application are displayed. These contain: *Owner*, *Pet*, *Visit*, *Vet*, *PetType*, *Speciality*, *VisitDate* and *BirthDate*.

Owner represents the owner of the pets, *Pet* represents the pet itself, *Vist* are the visits done by the pets in the clinic, *Vet* represents the vets available in the clinic, *PetType* represents the type of pet each pet is, for example a dog, a cat or a bird, *Speciality* represents the specialities of the vets, for example radiology, surgery, dentistry, *VisitDate* represents the date of a visit of the pet and *BirthDate* is the birth date of the pet.

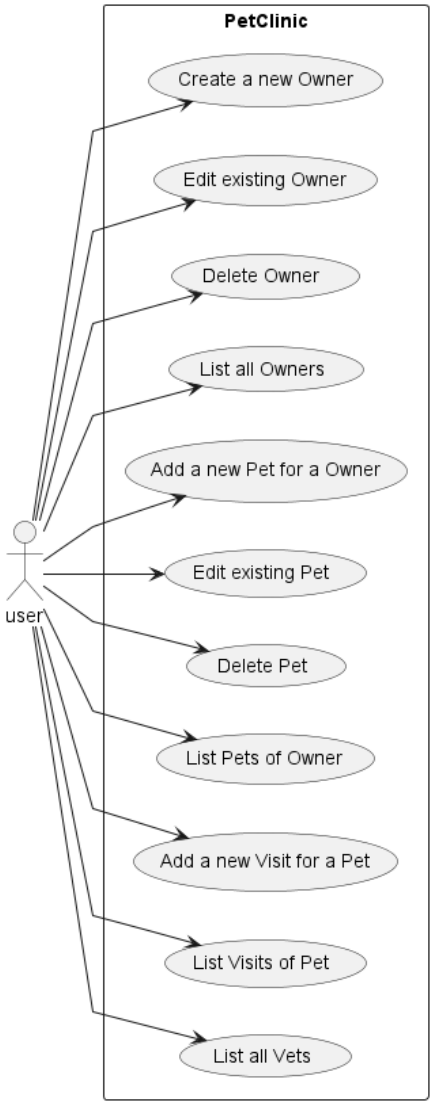


Figure 6 - Spring PetClinic Application Use Cases
Adapted From: [77]

The use cases available for the Spring PetClinic Application are listed in the Use Case Diagram in Figure 6. These Include:

- Create a new Owner – This use case allows the user to create a new owner in the system.
- Edit existing Owner – After creating owners, they are made available in the system. The user can edit these owners.
- Delete Owner – The user can delete existing owners registered in the system.
- List all Owners – The user can list all the owners available in the system.
- Add a new Pet for an Owner – The user can insert new pets and associate them with their owners.
- Edit existing Pet – The user can edit pets that are available in the system.
- Delete Pet – The user can delete existing pets.
- List Pets of Owner – Opposite to the owners, the user can list the pets of their owner and not all of them together.
- Add a new Visit for a Pet – The user can add visits for different pets.
- List Visits of Pet – After adding the visits to the pet, the user can list them.
- List all Vets – The user can list all the vets and see their information available on the system.

These are the functionalities the Spring PetClinic Application offers to the user.

5.1.2.2 Architecture

As mentioned, the Spring PetClinic project is community-driven. It has been built in several different architectures. For this work, the most relevant are the non-modular monolithic, modular monolithic and microservices.

In the Figure 7, using domain-driven design, the business concepts from the domain model are shown within their aggregates.

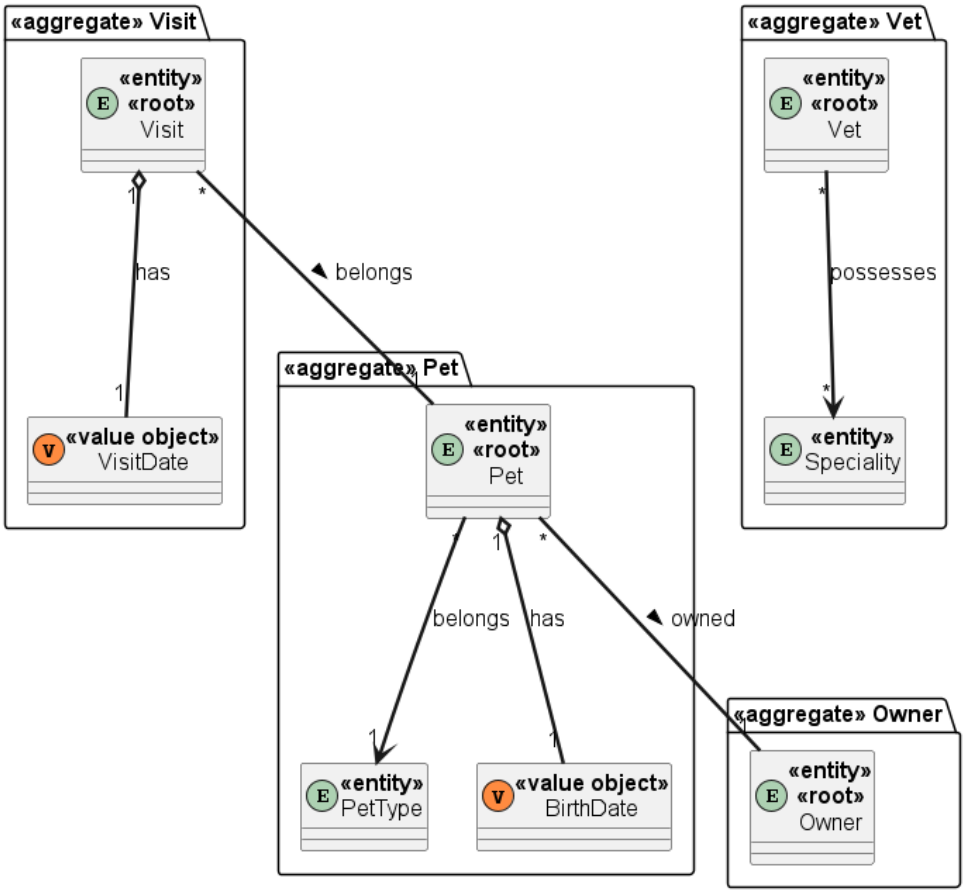


Figure 7 – Spring PetClinic Application Aggregates
Adapted From: [77]

The aggregates of the application are shown in Figure 7. In this system, there are 4 aggregates, these are:

- Visit – This aggregate represents the visits to the PetClinic of the pets, having their date associated.
- Pet – This aggregate contains all the information about the Pet, including its type and birth date. It is also connected to the Owner, since it has an owner.
- Owner – This aggregate represents the owner and their information.
- Vet – This aggregate represents the vets of the veterinary clinic and their speciality.

In the following subsections, the way the application is composed is explained for the non-modular monolithic, modular monolithic and microservices. Each type of architecture is composed differently, so for each one of them, the way the application is structured differs.

5.1.2.2.1 Non-Modular Monolithic

First, the non-modular monolithic structure is analysed.

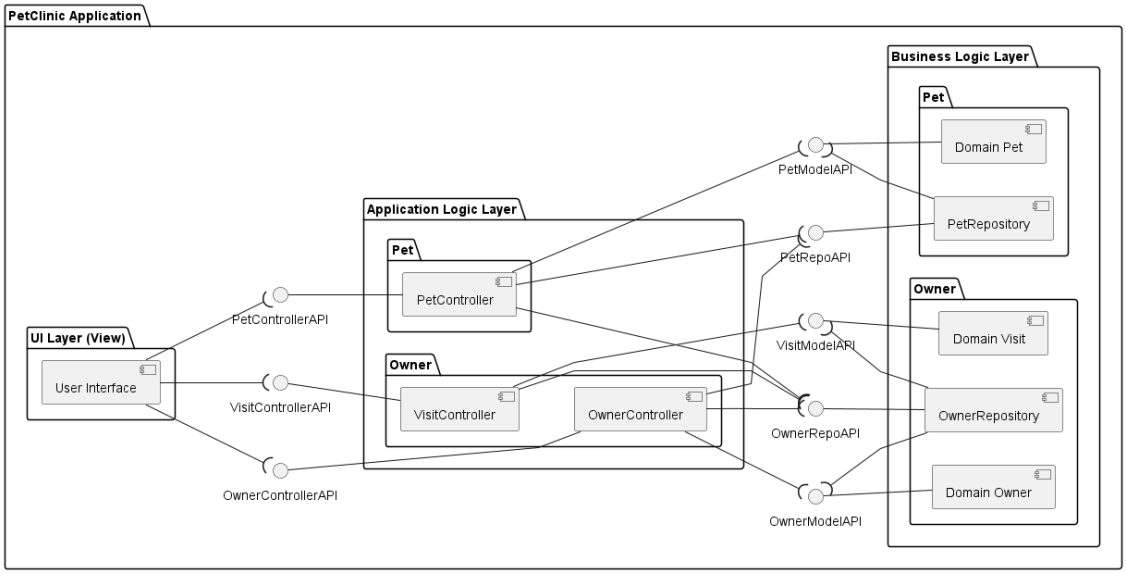


Figure 8 – Component Diagram Level 3 Spring PetClinic Application Non-Modular Monolithic
Source: [76]

Figure 8 shows the component diagram (Logical View) of the non-modular monolithic version Spring PetClinic Project. This version is built in only one codebase. It contains three layers, and each layer has components from each of the aggregates. However, as this is the base application, the Visit and Owner entities are packed together on the same aggregate.

The first layer (from the left to the right) is the User Interface (UI) Layer. This layer is used by the user to interact with the application and presents all the graphical content needed for the functionalities. It consumes the interfaces made available by the three controllers.

The second layer is the Application Logic Layer. This layer contains the three controllers for each entity. Each one of them provides its interface for the UI Layer to consume. However, because there are entities that need to get information about others (check Figure 7), these controllers consume from interfaces other than those of the same aggregate. The *PetController* needs to consume from the *OwnerReporitory* interface; similarly, the *VisitController* also needs to consume from the same interface. The *OwnerController* consumes from the *PetRepository* interface.

The third layer is the Business Logic Layer. It contains the domain entity and the respective repositories. The Visit Entity is the only one that does not have a repository since it is directly associated with the owner in this architecture.

In this architecture, there is only one database; consequently, all the data is persisted in the same database.

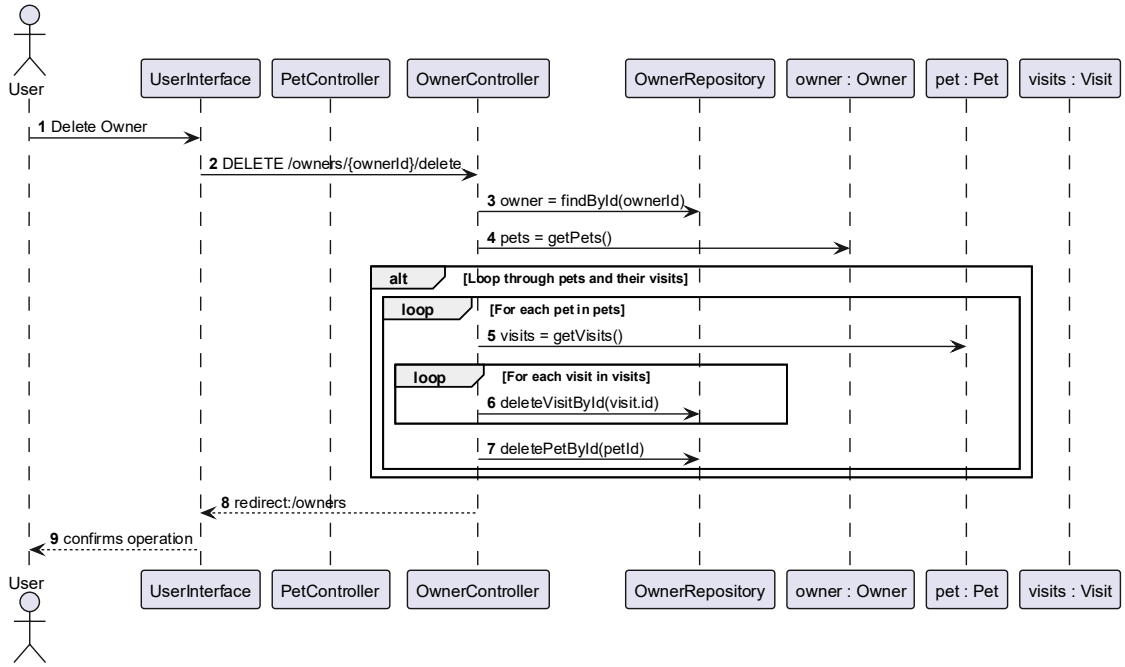


Figure 9 – Delete Owner Sequence Diagram non-modular Monolithic
Source: [76]

The use case Delete Owner is demonstrated in a sequence diagram in Figure 9. This sequence diagram makes it possible to visualise how the components interact with each other in this architecture. The Actor communicates to the User Interface, which sends a request to the controller, then the controller, containing the domain entities, finds the owner, finds the pets from the repositories, and then loops it, deleting the visits for each pet and also each pet from the owner, finally redirecting the user to the success page that confirms the operation.

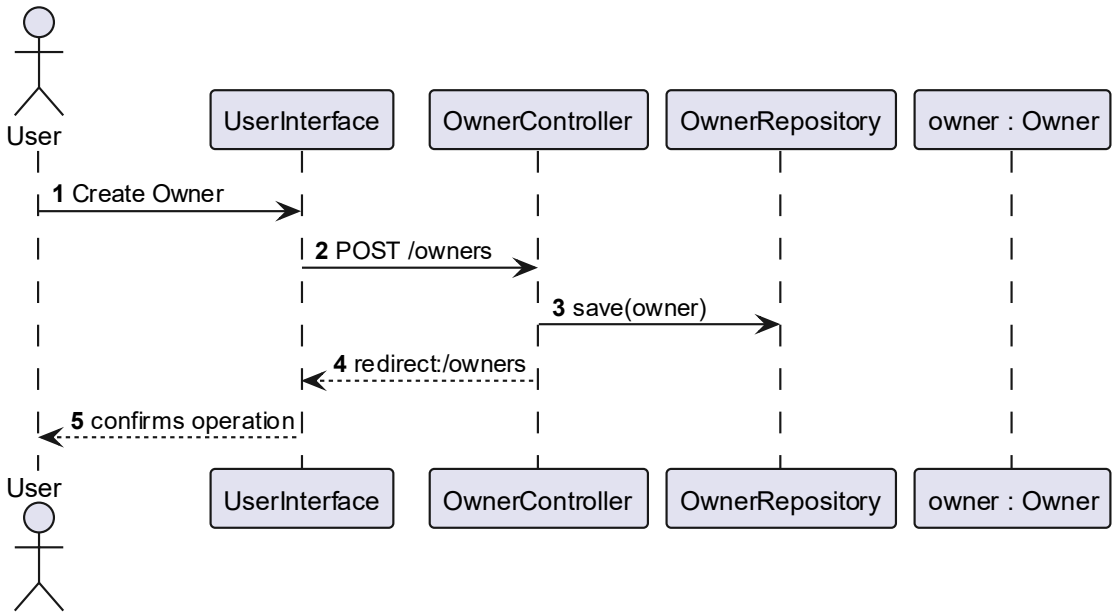


Figure 10 - Create Owner Sequence Diagram non-modular Monolithic
Source: [76]

Figure 10 shows the sequence diagram for the Create Owner use case. In this sequence diagram, there is a request from the user to create the owner, followed by the persistence of the data, resulting in success.

5.1.2.2.2 Modular Monolithic

As a modular monolithic system, the components need to be separated into well-defined modules. For this reason, the community decided that the Visit Domain Entity should evolve into a new module when migrating from a non-modular to a modular monolithic.

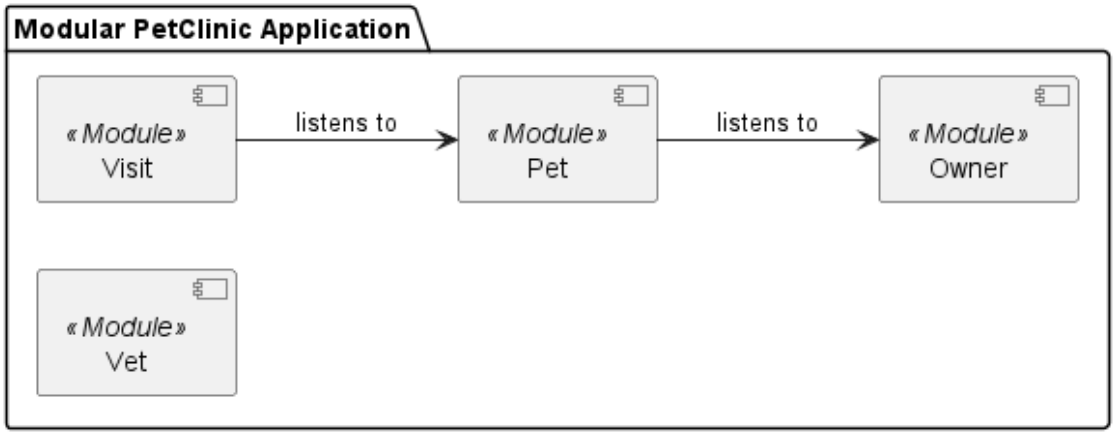


Figure 11 – Modules Communication Modular Monolithic Spring PetClinic Application
Source: [77]

The modules defined for the modular monolithic are the same as the aggregates: Visit, Pet, Owner and Vet. These modules communicate with each other through domain events, removing the dependency between them. These modules are represented in Figure 11.

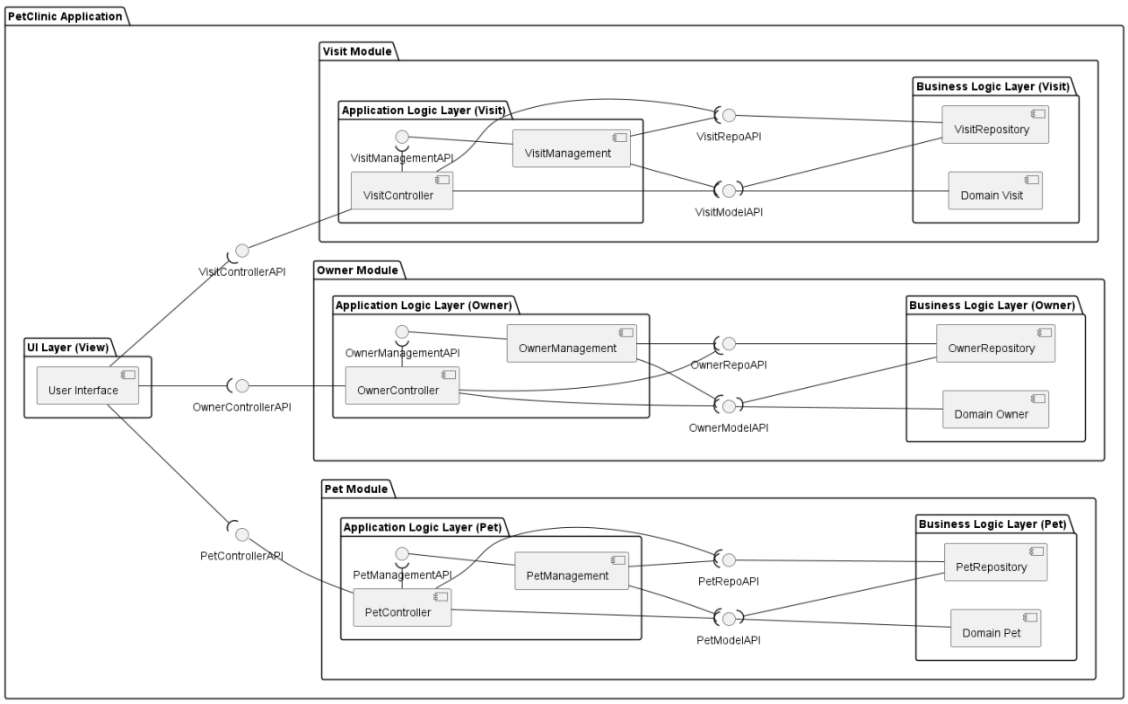


Figure 12 – Component Diagram Level 3 Spring PetClinic Application Modular Monolithic
Source: [77]

Similarly to Figure 8, Figure 12 shows the component diagram level 3 of the Spring PetClinic Project for the Modular Monolithic application. In this case, the components are divided into their modules and then by the layers, excluding the UI Layer, since it does not contain any business logic.

In this component diagram, it is relevant to note that there are no more interface consumptions of different modules, since there cannot be any dependencies between modules. After all, the interactions are made by domain events, and each module listens to the relevant events for them and acts on them. This is very similar to what happens in a microservices architecture; the difference is that this is all in one whole application. The new components created – Management – manage the domain events that affect the module in which they are.

The visit module has its repository; apart from that, everything is composed equally to the non-modular monolithic version.

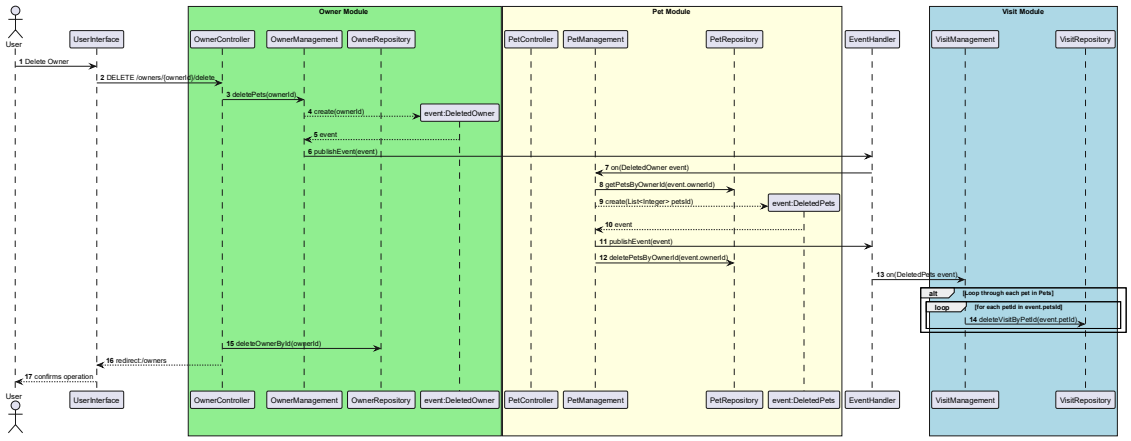


Figure 13 – Delete Owner Sequence Diagram, Modular Monolithic
Source: [76]

The sequence diagram for the use case Delete Owner on the modular monolithic is displayed on Figure 13. The difference between the non-modular monolithic (Figure 9) is that these are now divided into well-defined modules – the owner module represented by the green box, the pet module represented by the yellow box and the visit module represented by the blue box – which interact between them through domain events, in this case, the *DeletedOwner* and *DeletedPets* event. These events are created (sequence number 4 and 9) and published (sequence number 6 and 11), and after, these events are treated by the Pet Module and Visit Module, in the *PetManagement* and *VisitManagment* component, these call the *PetRepository* and *VisitRepository* respectively, and the last one needs to loop through each pet so that deletes every visit associated.

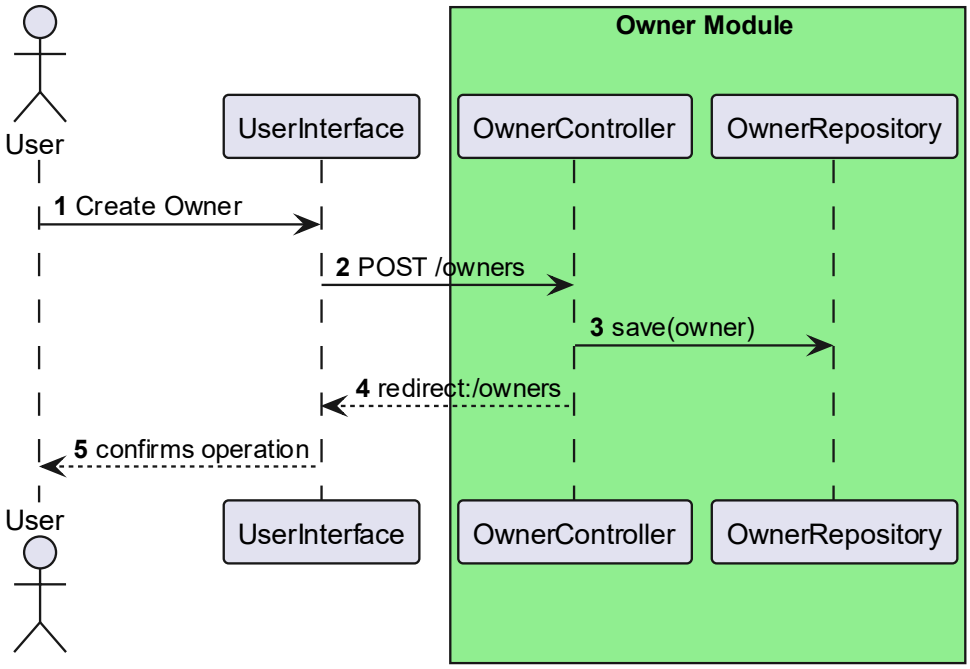


Figure 14 – Create Owner Sequence Diagram, Modular Monolithic
Source: [76]

Figure 14 represents the Create Owner use case Sequence Diagram. In this use case, only one module is affected. This use case starts with the user requesting to create the owner, followed by the persistence of the data needed, and responding with the confirmation of the operation.

5.1.2.2.3 Microservices

The microservices Spring PetClinic application is very similar to the modular monolithic application. As referenced in other chapters, the modular monolithic is a bridge step to migrate from a monolithic to this architecture.

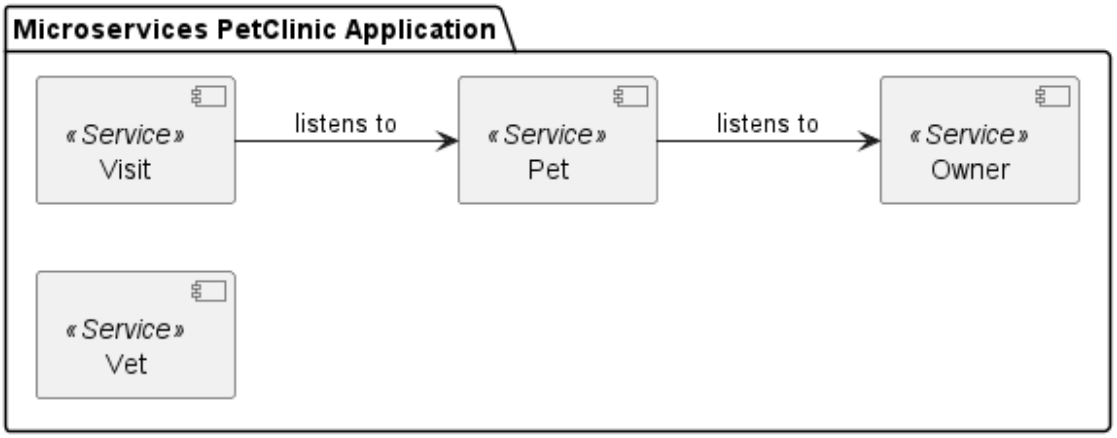


Figure 15 - Services Communication Microservices Spring PetClinic Application
Source: [76]

The service communication is graphically demonstrated in Figure 15. This figure is almost identical to Figure 11, which represents the module communication on the modular monolithic. The difference is that instead of modules, there are services, each one of which is independent, meaning that it can be deployed independently.

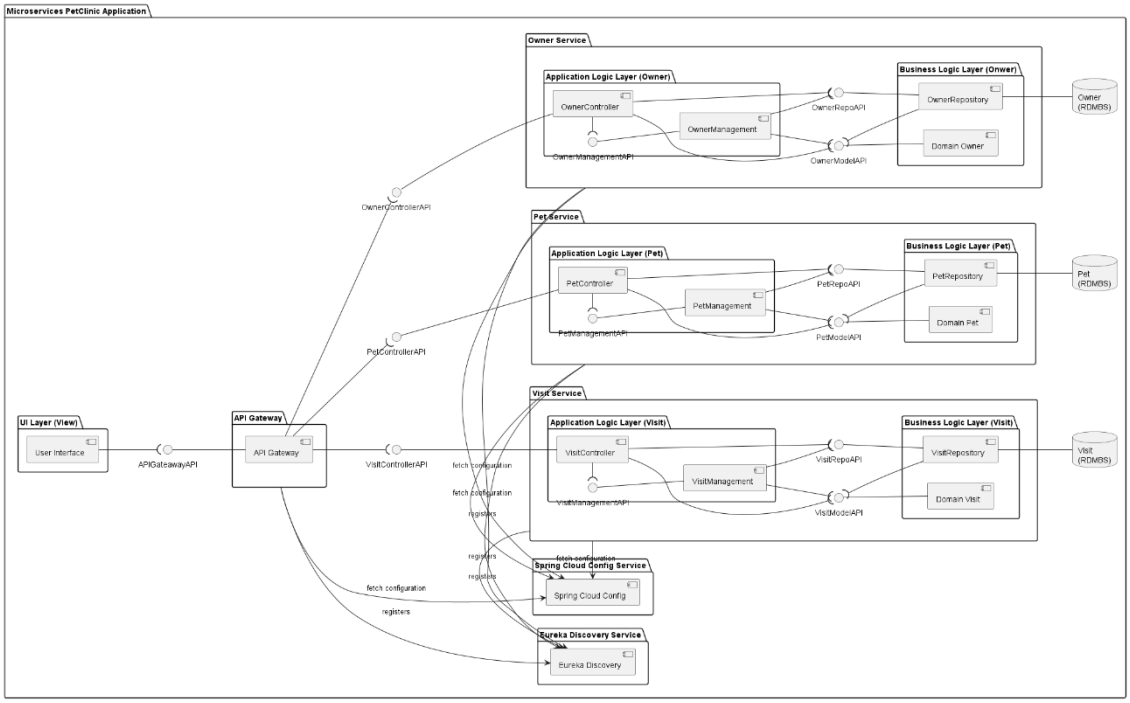


Figure 16 - Component Diagram Level 3 Spring PetClinic Application Microservices
Adapted From: [78]

The component diagram that represents the microservices version of the Spring PetClinic Project is shown in Figure 16. Comparing it with the modular monolithic version (Figure 12), instead of modules, there are independent services which are deployed independently, as mentioned. In this version of the application, each one of the services must be registered in the discovery service (Eureka Discovery Service), so that the API Gateway knows the path to each service. The configuration in this project is in another service (Spring Cloud Config Service), where the services fetch the configuration. Like the modular monolithic version, there are no dependencies between services, since the communication is also done by events, removing the coupling between services. Another difference, graphically represented on the diagram, is that each service has its own database; in contrast, the monolithic applications use one physical database per module/aggregate.

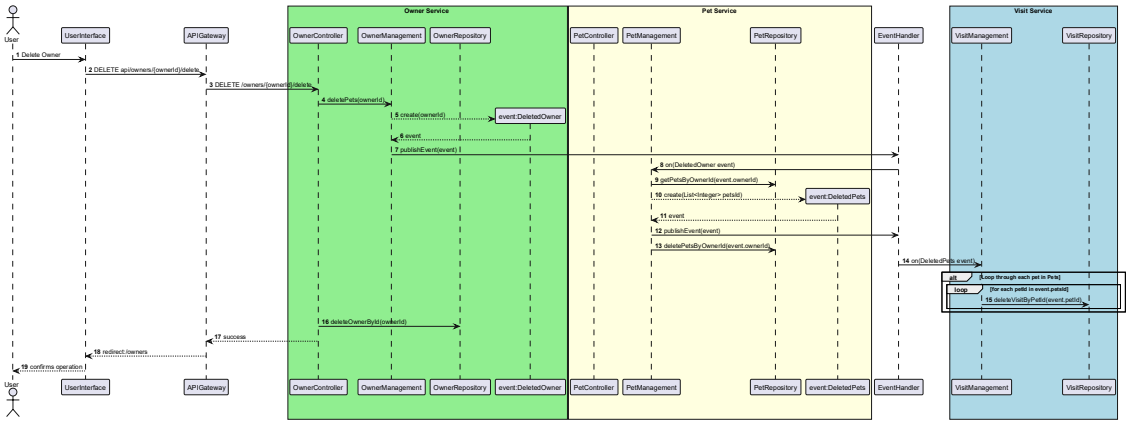


Figure 17 - Delete Owner Sequence Diagram Microservices
Source: [76]

The sequence diagram for the Delete Owner Use Case of the microservices is shown in Figure 17. This sequence diagram is also identical to the modular monolithic version (Figure 13), only differing by adding the API Gateway, which redirects requests to the proper service and instead of modules, there are the independent services which communicate with an event handler. The workflow of this use case is the same as the modular monolithic.

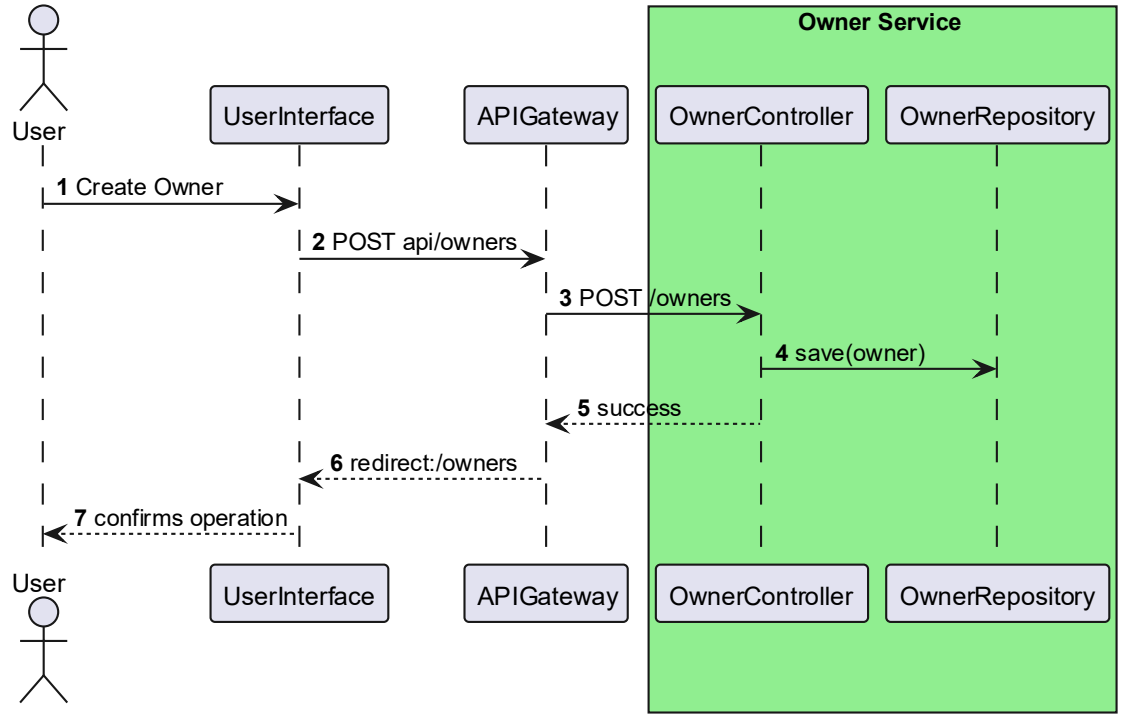


Figure 18 – Create Owner Sequence Diagram Microservices
Source: [76]

The Create Owner use case Sequence Diagram is represented in Figure 18. This use case, like the modular monolithic, only affects one service, including the API Gateway. This use case starts

with the request of the user to create a new owner, followed by the persistence of the owner data and finally, confirming the success of the operation.

5.2 Modifications

This chapter delves into the application's migration process to achieve the most transparent evaluation results.

As referenced in the section 5.1.2, the three application versions of the Spring PetClinic Project (non-modular monolithic, modular monolithic and microservices) have already been developed by the community. However, the non-modular and modular monolithic applications differ from the microservices application, lacking important functionalities.

The use cases whose microservice application lacks and, therefore, have been migrated are the Delete Owner and Delete Pet. These two functionalities are critical to be able to gather the best data, which are functionalities that promote the module and services communication. By having these two functionalities available between the three versions of the application, the data obtained has higher quality and therefore, the results are more reliable, directly affecting the outcome of this work.

5.2.1 Microservices Application Migration Process

In the following section, the migration process for the microservices application structure and functionalities implemented is addressed.

5.2.1.1 Application Structure Migration Process

In the following subsection. The restructuring of the modules is shown and explained. It aims to achieve coherence with the documentation, since it can directly affect the maintainability of the application.

Figure 19 – Service structure for Microservices Application before Migration
Source: [78]

Figure 19 shows how the services were structured before the migration. The *customers-service* had the *Owner* and *Pet* entity business logic, which is not per the documentation in Figure 15.

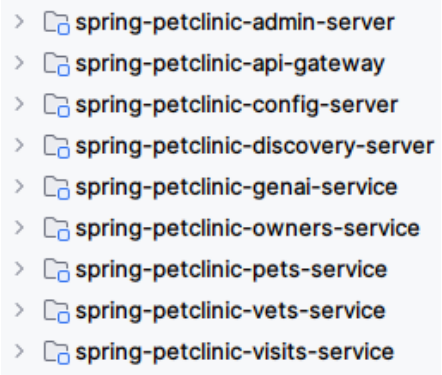


Figure 20 – Service structure after Migration for Microservices Application
Source: [76]

The *customers-service* is, therefore, separated into two different services: *owners-service* and *pets-service*, achieving accordance with the documentation represented in Figure 15.

Following the restructuring of the modules, the components were also not in line with the documentation in Figure 16. For this reason, the packages were restructured.

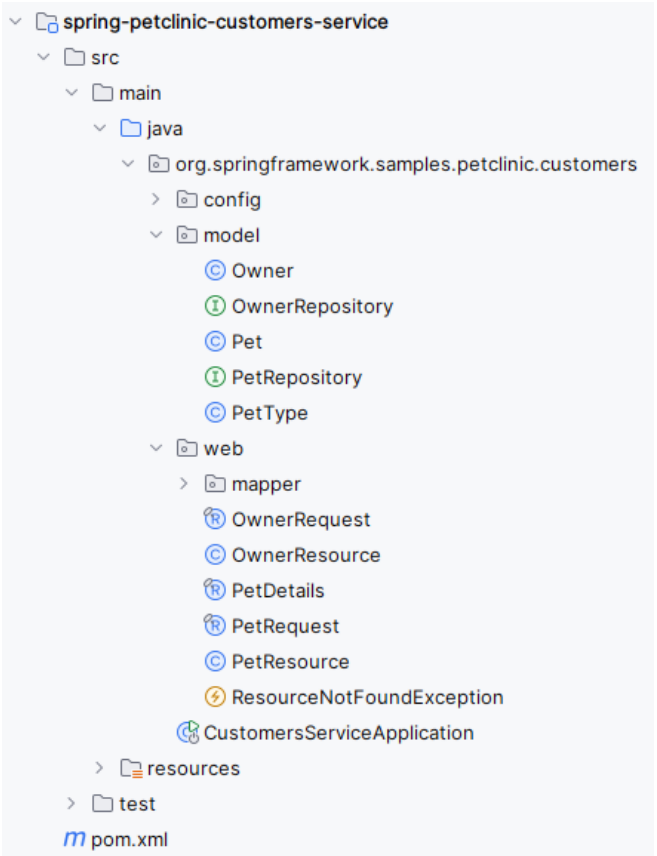


Figure 21 – Customers Service Package Structure for Microservices Application before Migration
Source: [78]

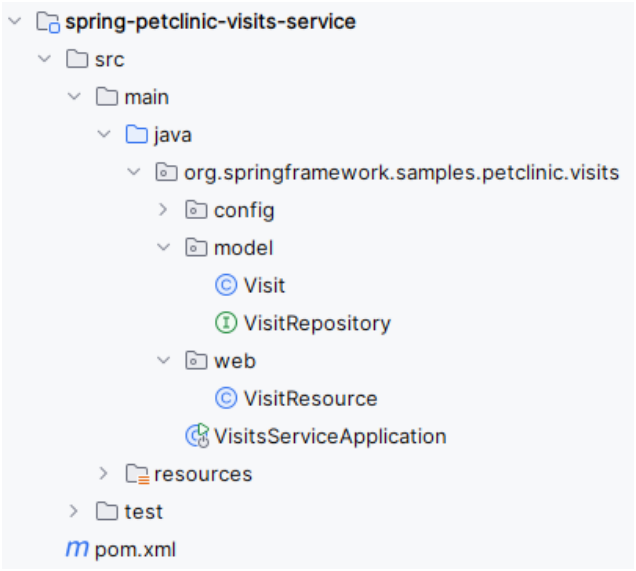


Figure 22 – Visits Service Package Structure for Microservices Application before Migration
Source: [78]

Figure 21 and Figure 22 show how the packages were structured for the *customers-service* and *visits-service* before the migration. There are two main packages, *model* and *web*. These packages do not represent how the application is structured in the modular monolithic version (Figure 12).

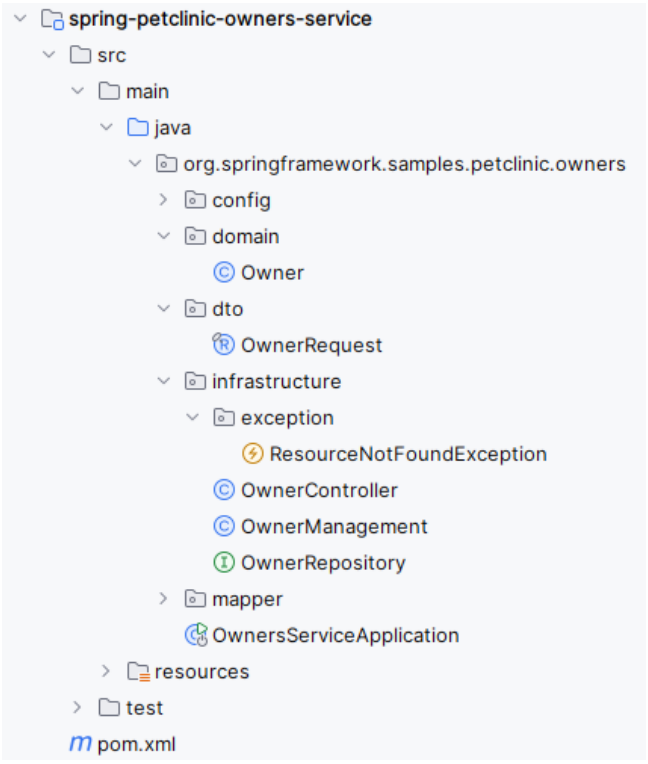


Figure 23 – Owners Service Package Structure for Microservices Application after Migration
Source: [76]

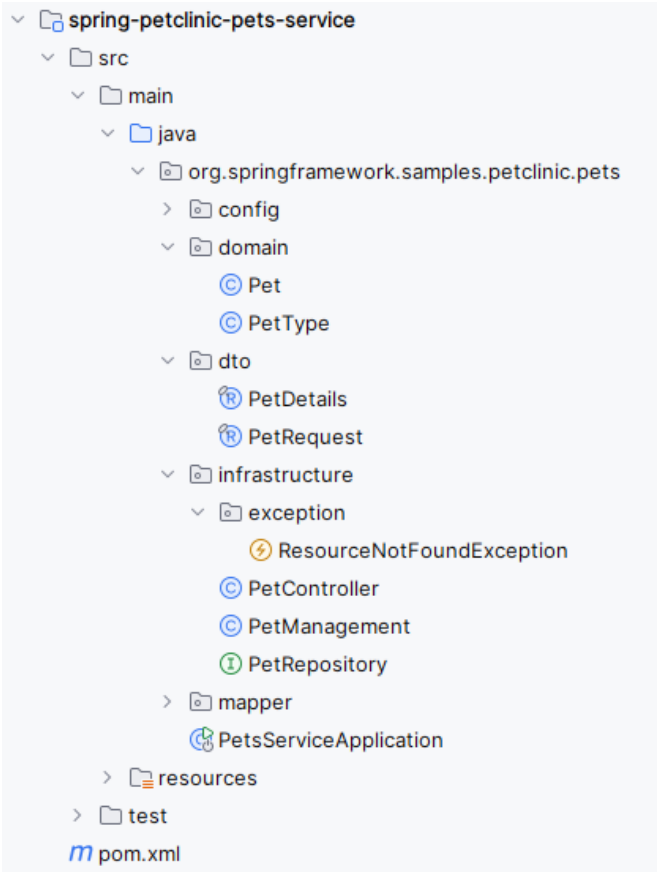


Figure 24 – Pets Service Package Structure for Microservices Application after Migration
Source: [76]

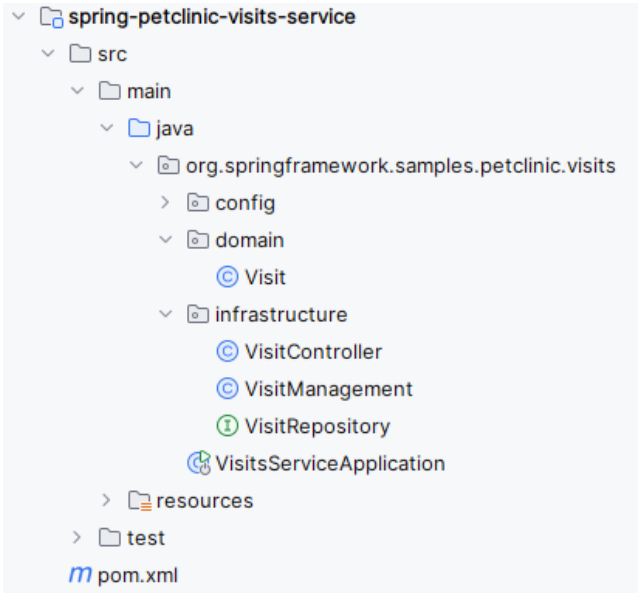


Figure 25 – Visits Service Package Structure for Microservices Application after Migration
Source: [76]

To implement the package structuration represented in the documentation (Figure 16), the structure was migrated to what Figure 23, Figure 24 and Figure 25 show. The Model and Web

packages were transformed into Domain and Infrastructure, achieving accordance with modular monolithic application structuration.

By migrating how the microservices application is structured, both with the number of services and the packages and component's structure, the evaluation will achieve more transparent data, given that the microservices represent a true migration process from modular monolithic to microservices.

5.2.1.2 Delete Owner Use Case Migration Process

To migrate the functionalities, the same approach as the modular monolithic is followed – Event Driven Design.

Based on Dobbelaere and Sheykh's study (2017) [79], comparing Kafka and RabbitMQ, the first one is more suited to scenarios in which the routing is simpler but has higher throughput demands, such as platforms for Big Data applications, that are based on data-later infrastructures to integrate batch and streaming services, fetching database change feeds and stateful stream processing. On the other hand, RabbitMQ is recommended for applications that require complex message routing, sophisticated filtering, request-response messaging, real-time operational metrics tracking, IoT application platforms, and for applications that require strict ordering mechanisms and fast and predictable message latencies.

Given the simple publish/consumer scenario for domain events and the information assed above, Kafka was the tool used to allow communication between independent services.

For the Delete Owner Use Case, a new endpoint was created, following the business logic needed.

```
@DeleteMapping(value = "/{ownerId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteOwner(@PathVariable("ownerId") @Min(1) int
ownerId) {
    ownerRepository.deleteById(ownerId);
    ownerManagement.sendOwnerDeleted(ownerId);
}
```

Code Snippet 1 – Delete Owner Endpoint

In the *owners-service*, inside the *OwnerController*, a new endpoint was created (Code Snippet 1). This endpoint represents Delete Request and it deletes the owner given by the Route, and after deleting it, it calls the function *sendOwnerDeleted* in the *OwnerManagement* class, sharing the *ownerId* which was just deleted.

```
@Component
public class OwnerManagement {

    KafkaTemplate<String, Integer> kafkaTemplate;
    private static final Logger log =
LoggerFactory.getLogger(OwnerManagement.class);

    public OwnerManagement(KafkaTemplate<String, Integer> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    void sendOwnerDeleted(final Integer ownerId) {
        CompletableFuture<SendResult<String, Integer>> future =
kafkaTemplate.send("ownerDeleted",ownerId);
        future.whenComplete((result, ex) -> {
            if (ex == null) {
                log.info("Sending message to Kafka Listener: {}", ownerId);
            }
            else {
                log.error("Failed to send message to Kafka Listener: {}",
ownerId, ex);
            }
        });
    }

}
```

Code Snippet 2 – *OwnerManagement* Class

In *OwnerManagement* class, the *KafkaTemplate* is injected, allowing to sending messages into the Kafka server, replicating the publishing of events.

To achieve a full asynchronous strategy, the event publication is done by a *CompletableFuture*, meaning it is resolved by threading. If it succeeds, it logs the info; on the other hand, if it fails, it logs the error. In this case, there is no business logic involved, although it could be implemented on this method.

After executing the deletion of the owner, there is now a message sent to the Kafka server, and it is resolved by the services listening to the *ownerDeleted* topic.

```
@Component
public class PetManagement {


    private final PetRepository petRepository;
    private final KafkaTemplate<String, Integer> kafkaTemplate;
    private static final Logger log = LoggerFactory.getLogger(PetControl-
ler.class);

    public PetManagement(PetRepository petRepository, KafkaTemplate<String,
Integer> kafkaTemplate) {
        this.petRepository = petRepository;
        this.kafkaTemplate = kafkaTemplate;
    }

    @KafkaListener(
        topics = "ownerDeleted",
        groupId = "kafka-group",
        containerFactory = "kafkaListenerContainerFactory"
    )
    @Transactional
    public void listenOwnerDeleted(Integer ownerId) {
        log.info("listenOwnerDeleted(ownerId={})", ownerId);
        List<Integer> petsId = petRepository.findByOwnerId(own-
erId).stream().map(Pet::getId).toList();
        petRepository.deleteByOwnerId(ownerId);
        for (Integer petId : petsId) {
            sendPetDeleted(petId);
        }
    }

    void sendPetDeleted(final Integer petId) {
        CompletableFuture<SendResult<String, Integer>> future = kafkaTem-
plate.send("petDeleted",petId);
        future.whenComplete((result, ex) -> {
            if (ex == null) {
                log.info("Sending message to Kafka Listener: {}", petId);
            }
            else {
                log.error("Failed to send message to Kafka Listener: {}",
petId, ex);
            }
        });
    }


}
```

Code Snippet 3 – *PetManagement* Class

Following what is documented in Figure 15, the *pets-service* listens to *owners-service*, meaning that after the *owners-service* publishes an event for an owner deletion, the *pets-service* listens to this event, as seen in the function *listenOwnerDeleted* in Code Snippet 3. Then it must get all the pets that need to be deleted and delete them, followed by publishing a new event *petDeleted* for each one of them, which is listened by the *visits-service*.

5.2.1.3 Delete Pet Use Case Migration Process

Similar, a new endpoint was created for the deletion of the owner, the same happens to the pet.

```
@DeleteMapping("owners/{ownerId}/pets/{petId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deletePet(@PathVariable("ownerId") Integer ownerId,
@PathVariable("petId") int petId) {
    petRepository.deleteById(petId);
    petManagement.sendPetDeleted(petId);
}
```

Code Snippet 4 – Delete Pet Endpoint

The Code Snippet 4 shows the new endpoint created in the *PetsController* that resides inside the *pets-service*. This endpoint is a Delete Request, which receives the *ownerId* and the *petId*. Currently, the *ownerId* does not hold any logic; however, the *petId* is needed to delete the pet itself.

After the deletion, the *sendPetDeleted* is called from the *PetManagement* class, already shown in Code Snippet 3. This method is the same as the listener for the *ownerDeleted* event, replicating the logic by sending the pet that has been deleted, publishing the *petDeleted* event into the Kafka Server.

```
@Component
public class VisitManagement {
    private final VisitRepository visitRepository;
    private final KafkaTemplate<String, Integer> kafkaTemplate;

    public VisitManagement(VisitRepository visitRepository,
KafkaTemplate<String, Integer> kafkaTemplate) {
        this.visitRepository = visitRepository;
        this.kafkaTemplate = kafkaTemplate;
    }


    @KafkaListener(
        topics = "petDeleted",
        groupId = "kafka-group",
        containerFactory = "kafkaListenerContainerFactory"
    )
    @Transactional
    public void listenPetDeleted(Integer petId) {
        visitRepository.deleteByPetId(petId);
    }


}
```

Code Snippet 5 – Visit Management Class

Following the documentation shown in Figure 15, the visits service listens to the *petDeleted* event published by the *pets-service*. The method that executes the logic is shown in Code

Snippet 5, *listenPetDeleted*. This method deletes the visits based on the *petId* received by the event.

By implementing Event Driven Design with Kafka, containing event publishers and listeners, it is possible to migrate both use cases implemented in the modular monolithic into a microservices architecture, developing asynchronous communication between independent methods, and not having dependencies between these services.

5.2.2 Adjustments

Since the architecture from the services changed, by decomposing the *customers-service* into *pets-service* and *owners-service*, there is a need to adjust the *apigateway-service*. This service is mainly configured by default using the *application.yml*, however, for aggregating information into the frontend application, some adjustments were made.

```
@GetMapping(value = "owners/{ownerId}")
public Mono<OwnerDetails> getOwnerDetails(final @PathVariable int ownerId)
{
    return customersServiceClient.getOwner(ownerId)
        .flatMap(owner ->
            visitsServiceClient.getVisitsForPets(owner.getPetIds())
                .transform(it -> {
                    ReactiveCircuitBreaker cb =
cbFactory.create("getOwnerDetails");
                    return cb.run(it, throwable -> emptyVisitsForPets());
                })
                .map(addVisitsToOwner(owner))
        );

}
```

Code Snippet 6 - Get all owner data API Gateway Endpoint after migration

In Code Snippet 6, a custom endpoint developed into the *apigateway-service* is shown, by making two requests, getting the information about the owner (method *getOwner* by the *CustomerServiceClient*), which it aggregates also the list of pets, before the migration, and then all the visits for each pet (method *getVisitsForPets* by the *VisitsServiceClient*).

```
@GetMapping(value = "owners/{ownerId}")
public Mono<OwnerDetails> getOwnerDetails(final @PathVariable int ownerId)
{
    Mono<OwnerDetails> result = ownersServiceClient.getOwner(ownerId)
        .flatMap(owner ->
            petsServiceClient.getPetsForOwner(ownerId).transform(it -> {
                    ReactiveCircuitBreaker cb =
cbFactory.create("getOwnerDetails");
                    return cb.run(it, throwable -> emptyPetsForOwner());
                })
                .map(addPetsToOwner(owner))
                .flatMap(pets ->
                    visitsServiceClient.getVisitsForPets(pets.getPetIds())
                        .transform(it -> {
                            ReactiveCircuitBreaker cb =
cbFactory.create("getOwnerDetails");
                            return cb.run(it, throwable ->
emptyVisitsForPets());
                        })
                        .map(addVisitsToOwner(owner))
                )

        );
    return result;
}
```

Code Snippet 7 - Get all owner data API Gateway Endpoint after migration

The migrated method now needs to do three requests, since there are three different services (*owners-service, pets-service* and *visits-service*).

The fetch of information about the owners resides now on *OwnersServiceClient*. After getting the owner's information, the information about the owner's pets is gathered by calling the *getPetsForOwner* method in *PetsServiceClient.* And finally, it calls the same method from the *VisitsServiceClient* as before, as seen in Code Snippet 7.

Another adjustment needed to improve the experiment's fairness was implementing an asynchronous strategy for publishing domain events in the modular monolithic application.

```java
package org.springframework.samples.petclinic;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.web.filter.HiddenHttpMethodFilter;

@Configuration
@EnableAsync
public class WebMvcConfig {

    @Bean
    public HiddenHttpMethodFilter hiddenHttpMethodFilter() {
       return new HiddenHttpMethodFilter();
    }

}
```

Code Snippet 8 – Notation in Configuration to allow asynchronous

The Spring Notation *@EnableAsync* allows the application to take advantage of asynchronous functionalities. This notation was added to the application configuration, as presented in Code Snippet 8.

5.2.3 Tests

To ensure the highest software quality and to follow the best code practices, tests have been developed for the new functionalities added to the microservices application. For each microservice that has had new functionalities added, namely the *owners-service*, *pets-service* and *visits service*, unit tests have been written. All the unit tests developed have their injected classes needed mocked, to allow the unit test to pass successfully.

```java
@Test
void shouldDeleteOwnerSuccessfully() throws Exception {
    int ownerId = 1;

    doNothing().when(ownerRepository).deleteById(ownerId);
    doNothing().when(ownerManagement).sendOwnerDeleted(ownerId);

    mvc.perform(delete("/owners/{ownerId}", ownerId)
            .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isNoContent());

    verify(ownerRepository).deleteById(ownerId);
    verify(ownerManagement).sendOwnerDeleted(ownerId);
}
```

Code Snippet 9 – Unit Test Owner Controller Delete Endpoint

Code Snippet 9 shows the unit test *shouldDeleteOwnerSuccessfully*. This unit test verifies if the endpoint created to allow users to delete owners, shown in Code Snippet 1, returns the right response status - no content, and if the *deleteById* method from *OwnerRepository* class and

sendOwnerDeleted method from *OwnerManagement* are called. Both of these methods are mocked to simulate the successful returns, in this case, doing nothing.

```
@Test
void sendOwnerDeletedShouldSendKafkaMessage() {
    Integer ownerId = 10;

    CompletableFuture<SendResult<String, Integer>> futureMock = Completa-
bleFuture.completedFuture(null);
    when(kafkaTemplate.send(eq("ownerDeleted"), any(Integer.class))).then-
Return(futureMock);

    ownerManagement.sendOwnerDeleted(ownerId);

    verify(kafkaTemplate, times(1)).send(eq("ownerDeleted"), ownerIdCap-
tor.capture());

    Integer capturedOwnerId = ownerIdCaptor.getValue();
    assert capturedOwnerId.equals(ownerId);
}
```

Code Snippet 10 – Unit Test Owner Management Publish Event Owner Deleted Method

To test the method *sendOwnerDeleted* from the *OwnerManagement* class, the test *sendOwnerDeletedShouldSendKafkaMessage* is shown in Code Snippet 10 has been implemented. This unit test mocks the completable future for the asynchronous event publication. Then, it verifies if the event has been published to Kafka, capturing the *ownerId* sent and verifies if both are equal.

```
@Test
void shouldDeletePetSuccessfully() throws Exception {
    int ownerId = 2;
    int petId = 2;

    doNothing().when(petRepository).deleteById(petId);
    doNothing().when(petManagement).sendPetDeleted(petId);

    mvc.perform(delete("/owners/{ownerId}/pets/{petId}", ownerId, petId)
            .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isNoContent());

    verify(petRepository).deleteById(petId);
    verify(petManagement).sendPetDeleted(petId);
}
```

Code Snippet 11 – Unit Test Pet Controller Delete Endpoint

Like what has been done for the owner deletion, the unit test *shouldDeletePetSuccesfully* shown in Code Snippet 11 has been implemented. It aims to verify if the delete request for the pet performs correctly, by verifying if the *deleteById method from PetRepository* and the *sendPetDeleted* method from the *PetManegement* class are called.

```
@Test
void sendPetDeletedShouldSendKafkaMessage() {
    Integer petId = 1;
    CompletableFuture<SendResult<String, Integer>> futureMock =
CompletableFuture.completedFuture(null);
    when(kafkaTemplate.send(eq("petDeleted"),
any(Integer.class))).thenReturn(futureMock);

    petManagement.sendPetDeleted(petId);
    verify(kafkaTemplate, times(1)).send(eq("petDeleted"),
petIdCaptor.capture());


    verify(kafkaTemplate).send("petDeleted", petId);

    Integer capturedPetId = petIdCaptor.getValue();
    assert capturedPetId.equals(petId);
}
```

Code Snippet 12 - Unit Test Pet Management Publish Event Pet Deleted Method

To verify if the event is being published correctly, the *sendPetDeleteShouldSendKafkaMessage* unit test has been implemented, as seen in Code Snippet 12, like what has been done for the owner in Code Snippet 10. It repeats the same steps as the one mentioned.

```
@Test
void listenOwnerDeletedShouldDeletePetsAndSendEvents() throws
ParseException {
    Integer ownerId = 1;
    SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy",
Locale.ENGLISH);
    //Pet 1
    Pet pet1 = new Pet();
    PetType petType1 = new PetType();
    petType1.setId(1);
    petType1.setName("Cat");
    pet1.setId(1);
    pet1.setOwnerId(ownerId);
    pet1.setName("Jeremias");
    pet1.setBirthDate(formatter.parse("10-10-2010"));
    pet1.setType(petType1);

    //Pet 2
    Pet pet2 = new Pet();
    PetType petType2 = new PetType();
    petType2.setId(1);
    petType2.setName("Dog");
    pet2.setId(2);
    pet2.setOwnerId(ownerId);
    pet2.setName("Alfredo");
    pet2.setBirthDate(formatter.parse("11-11-2011"));
    pet2.setType(petType2);

    List<Pet> pets = List.of(pet1,pet2);

    when(petRepository.findByOwnerId(ownerId)).thenReturn(pets);
    doNothing().when(petRepository).deleteByOwnerId(ownerId);

    CompletableFuture futureMock = mock(CompletableFuture.class);
    when(kafkaTemplate.send(eq("petDeleted"),
any(Integer.class))).thenReturn(futureMock);

    petManagement.listenOwnerDeleted(ownerId);

    verify(petRepository).deleteByOwnerId(ownerId);
    verify(kafkaTemplate, times(2)).send(eq("petDeleted"),
petIdCaptor.capture());

    List<Integer> capturedIds = petIdCaptor.getAllValues();
    assert capturedIds.contains(1);
    assert capturedIds.contains(2);
}
```

Code Snippet 13 – Unit Test Pet Management Listen Pet Owner Deleted Method

Additionally, whenever an owner is deleted, the pets service must actively listen for the event published to delete the pets and then publish an event notifying that a pet has been deleted. The unit test *listenOwnerDeletedShouldDeletePetsAndSendEvents* has been implemented to verify the entire workflow explained, as demonstrated in Code Snippet 13.

Two pets are mocked into the repository, and when the Kafka server calls *listenOwnerDeleted*, the repository returns them. Then, these pets are deleted, and two events must be published.

```java
@Test
void listenPetDeletedShouldDeleteVisits() {
    Integer petId = 5;
    doNothing().when(visitRepository).deleteByPetId(petId);
    visitManagement.listenPetDeleted(petId);
    verify(visitRepository).deleteByPetId(petId);
}
```

Code Snippet 14 - Unit Test Visit Management Listen Event Pet Deleted Method

Finally, in the Code Snippet 14 it is shown the *listenPetDeletedShouldDeleteVisits* unit test that verifies that after the *visits-service* listens for the pet deleted event, the visits associated with that pet are also deleted.

Integration tests were also developed using Postman. These integration tests verify if all the workflows work as intended, without verifying independently the classes and without mocks.

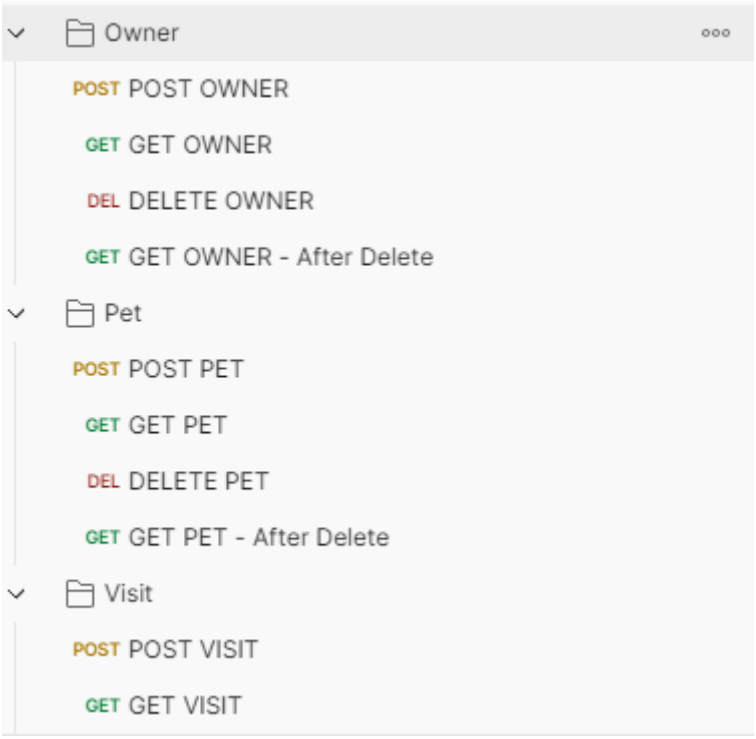


Figure 26 – Integration tests for Owner, Pet and Visit Postman

Taking advantage of collection variables, by saving the return data from the endpoints, and using the JavaScript available to store these values and test them, after creating each folder for each entity, as seen in Figure 26.

For the owner folder, first, an owner is created, then it is tested the GET Request of that owner, returning the owner just created, followed by the deletion and then verifying that it does not exist anymore. The same is done for the Pet entity and the visit. For the visit entity, there is only the creation and the verification of whether it exists.

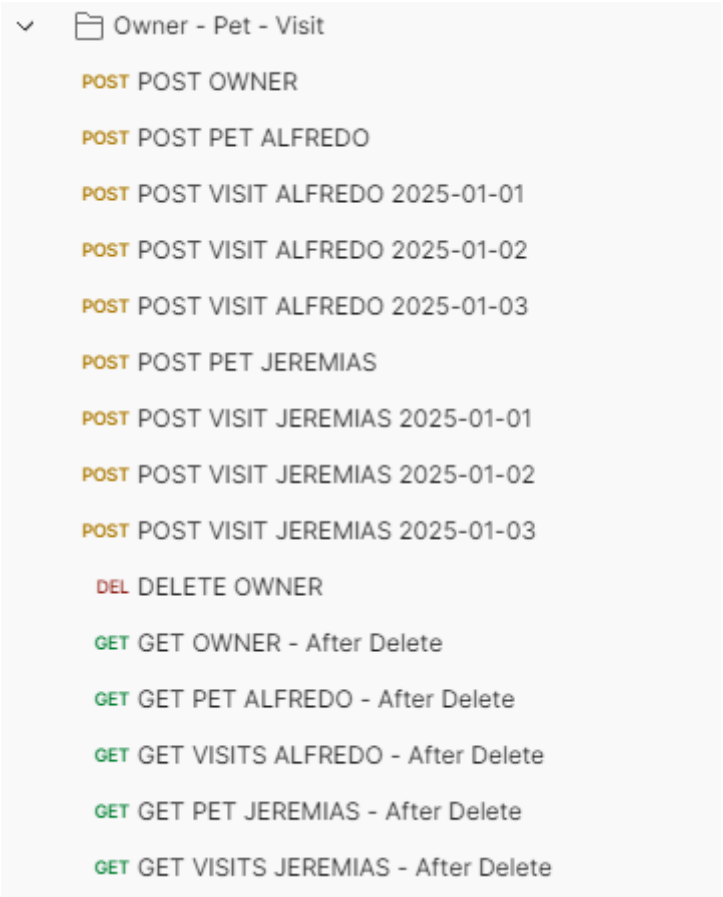


Figure 27 – Integration tests for Owner deletion with pets and visits verification

The final test, as seen in Figure 27, combines the full workflow developed using the event-driven design. It first creates all the entities needed, an owner, two pets and their correspondent visits. Then it only deletes the owner, to verify if all the data associated has also been correctly deleted. This ensures that all the newly developed functionalities and events are working as intended.

The unit and integration tests developed help to ensure the quality of the software, confirming the application is working as intended. This adds value to the experiment being conducted because it proves that the application is working and functioning as intended.

5.2.4 Docker Deployment

To achieve the most transparent and coherent results for the experiment being developed, limitations must be applied to the application when deployed and analysed. For this reason, the deployment used to evaluate the applications was made by containers with resource limitations.

The tool used to create containers, including monolithic applications and all the services from the microservices application, was Docker. Even though all the projects used Docker for their deployment, the monolithic versions used the Spring Boot plugin, which did not comply with what was intended. Because of this, a Docker image configuration and *docker-compose*

configuration file were written. In the case of the microservices application, it already had the Docker images ready; it just needed to adjust the *docker-compose* file with the limitations and migrate the *customers-service* into the two services, *owners-service* and *pets-service*.

The following process is followed in each application to ensure that the Docker deployment is done successfully:

- Generate Artefacts – The Docker containers run the application compiled to replicate a production scenario.
- Build the Docker images – Using the *dockerfile* configuration, the Docker images are built based on the artefacts.
- Docker Compose: With the docker-compose file, the configurations for resource limitations were added, based on the images built, containers are therefore created and ready to use, simulating a production environment.

```
FROM eclipse-temurin:17-jdk-alpine

WORKDIR /app

COPY ../target/spring-petclinic-*.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Code Snippet 15 – Dockerfile for Non-modular and modular monolithic application

In Code Snippet 15, the *dockerfile* configuration for both non-modular and modular monolithic is shown. As referenced, after building the artefacts, the compiled application builds the Docker images.

```yaml
name: petclinic_monolith

services:
  petclinic:
    image: petclinic-monolith
    container_name: petclinic-monolith
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
        reservations:
          cpus: 0.5
          memory: 256M
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - SPRING_PROFILES_ACTIVE=h2
    restart: unless-stopped

  mysql:
    image: mysql:8.2
    ports:
      - "3306:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=
      - MYSQL_ALLOW_EMPTY_PASSWORD=true
      - MYSQL_USER=petclinic
      - MYSQL_PASSWORD=petclinic
      - MYSQL_DATABASE=petclinic
    volumes:
      - "./conf.d:/etc/mysql/conf.d:ro"
    profiles:
      - mysql
  postgres:
    image: postgres:16.1
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_PASSWORD=petclinic
      - POSTGRES_USER=petclinic
      - POSTGRES_DB=petclinic
    profiles:
      - postgres
```

Code Snippet 16 – Docker Compose file for the non-modular application

To deploy the monolithic applications, a *docker-compose.yml* file has been written, available in Code Snippet 16. A replica of this file was also written for the modular monolithic application, changing only the name, image and container name. Even though this file has available Postgres and MySQL containers for data persistence, the in-memory H2 database was used, solely to focus on the evaluation of maintainability, performance and energy efficiency of the application

and not include other components. The containers are limited to 1 CPU and 512 MB of memory to ensure the validity of the experiment.

The *docker-compose.yml* file for the microservices application version is presented in Appendix B

To ensure a fair comparison between the systems, each architecture was assigned equivalent resource limits and reservations. Since the monolithic applications run as a single container, both were limited to 1 CPU and 512 MB of memory, with reservations set at 0.5 CPUs and 256 MB. For the microservices, these resources were assigned to each core business service (*owners-service*, *pets-service*, *visits-service*, and *vets-service*), meaning each service individually received the same limits and reservations as the monolithic. Additionally, supporting services without business logic—such as *config-server*, *discovery-server*, *kafka-server*, and *api-gateway* - also received their resource allocations, each limited to 1 CPU and 512 MB of memory. This inherently means that the microservices architecture collectively has access to more resources overall, a factor to consider when evaluating the results.

5.3 Summary

Summarising this section, after defining the process for the project selection and analysing the project itself – Spring PetClinic - in terms of business context and architecture, choosing the right application with appropriate criteria is crucial to conduct a reliable and transparent controlled experiment. The project chosen meets the necessary characteristics, including being open-source, having a license that allows the use, distribution and modification, object-oriented programming and domain-driven-design principles, and being available in different architectural designs – non-modular monolithic, modular monolithic, using Spring Modulith, and microservices.

The analysis of the design for each architecture shows the differences between each architecture, both structural and functional. The microservices architecture required adjustments and additional migration, with the help of Kafka and event-driven design, to align with the modular monolithic structure and ensure that all the functionalities available in the modular monolithic are also available in the microservices application, aiming to achieve higher transparency and accuracy for the controlled experiment.

Tests were conducted, including unit and integration tests, and controlled Docker deployment environments, so that the experiment achieves the highest quality possible and removes variables that can alter the results. Overall, this approach assures that the results gathered solely focus on the impacts of modularisation on performance, maintainability and energy efficiency, and nothing more.

Finally, one of the most critical considerations for the experiment is a fair comparison between the three architectural styles, and at the same time, respecting the nature and constraints of

each architectural style. A particular point is the distribution of computational resources across the different deployments.

In the case of monolithic architectures, the entire application runs as a single container. Therefore, resource limits and reservations were defined at the container level—specifically, 1 CPU and 512 MB of memory, with reservations of 0.5 CPUs and 256 MB. For the microservices application, equivalent resource limits were assigned per business service, namely owners-service, *pets-service*, *visits-service*, and *vets-service*. Supporting services such as *config-server*, *discovery-server*, *kafka-server*, and *api-gateway* also received their dedicated resource allocations. This setup means that, in aggregate, the microservices deployment has access to more computational resources than the monolithic alternatives.

While this may initially appear to compromise the fairness of the comparison, this decision was a deliberate and necessary design trade-off. Each microservice is a standalone application that must independently manage its runtime environment, including its own Spring context, database connection pool, messaging client (Kafka), and HTTP server. These layers introduce significant runtime overhead that monolithic systems, especially the non-modular variant, do not incur. Assigning equal total resources to a microservices deployment as to a monolithic would artificially constrain the architecture and fail to reflect the conditions under which microservices are typically expected to operate.

6 Results

This chapter outlines the methodology used to analyse the effects of modularisation within a software application. The primary aim is to evaluate how modularisation influences performance, maintainability, and energy efficiency by comparing three architectural approaches: non-modular monolithic, modular monolithic, and microservices. Section 6.1 introduces the Goal/Question/Metric (GQM) framework, which is used to formalise the objectives of the analysis. section 6.2 then presents and discusses the results of the conducted tests. The chapter concludes with Section 6.3, which summarises the key findings and reflections from the analysis.s

6.1 Methodology

The Goal/Question/Metric Method is a framework first used in the NASA Goddard Space Flight Center. This framework aims to make measurements goal-oriented. Therefore, the company/project should define the goals it wants to achieve. Then, questions are refined based on the goal, and metrics are provided that should give the information to answer these questions. The GQM Method defines the metrics from a top-down perspective; on the other hand, the data is interpreted from a bottom-up perspective [80]. In Figure 28 is possible to visualise the structure that allows GQM to be defined by a top-down perspective and interpreted from a bottom-up perspective.

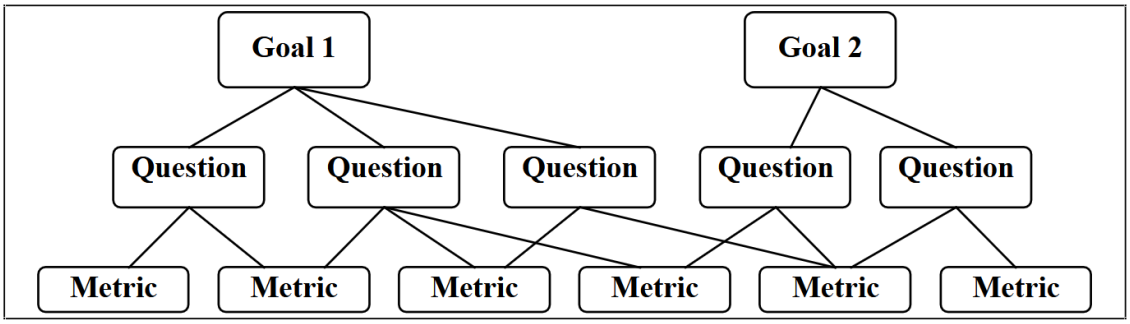


Figure 28 – Visualisation of Goal/Question/Metric
Source: [81]

The GQM has, therefore, three levels. Each one of them is related to the three main components [81]:

1. Conceptual Level (Goal) – A goal is defined based on an objective, depending on the environment, quality models and other factors.

2. Operational Level (Question) – A question describes how the goal needs to be evaluated. Depending on the quality issue and point of view, they define what is being measured.

3. Quantitative Level (Metric) – Metrics are quantifiable measures that answer the questions.

The Table 3 describes the GQM method applied. In the first column, the goal is referenced, followed by the questions and metrics.

Table 3 – Goal, Questions and Metrics

Goal	Questions	Metrics
Analyse the effects of modularisation on software applications.	How does modularisation affect the performance of a software application?	Response Time
		Throughput
	How does modularisation affect the maintainability of a software application?	Average Complexity
		Average Component Dependency
		Normalised Cumulative Component Dependency
		Propagation Cost
		Energy Consumption
	How does modularisation affect the energy efficiency of a software application?	CPU Cycles per Task
		Energy Consumed per CPU cycle
		CPU cycles per Energy Consumption

The metrics mentioned above can be gathered manually or using tools. The following sections explain each metric and reference the method used.

6.1.1 Performance

To analyse how modularisation can affect performance, the chosen metrics are Response Time and Throughput.

Response time, as the name suggests, is the time that the response takes. Meaning, it is the time recorded from the moment the user requests until the user receives the full response from the server. This metric is crucial since it is directly related to the user experience, and it gives an understanding of how fast an application can react to requests, affecting overall performance. Referenced in Section 3.1.2.1, the tool Grafana K6 is used to gather this metric.

Throughput is related to response time because it is the rate at which an application can complete tasks given a timeframe. It can be measured by bits per second or requests per second. Related to the response time, it is possible to measure the throughput of an application given how many requests it can handle per second. This metric directly reflects an application's ability to handle workload efficiently. Higher throughput generally indicates better performance, as it means the system can handle more work within a given period. Like Response Time, Grafana K6 is used to gather this metric.

Table 4 shows the summary of the metrics being gathered regarding performance.

Table 4 – Performance metrics summary

Metric	Unit	High Values are	Best Value	Worst Value	Metric Summary
Response Time	Milliseconds (ms)	Negative	**Lowest**	**Highest**	Time between request initiation and receiving the full response. Lower values improve user experience.
Throughput	Requests/s	Positive	**Highest**	**Lowest**	The number of requests handled per second is higher, which means better system capacity and efficiency.

6.1.2 Maintainability

To measure how modularisation affects maintainability, the metrics Average Complexity, Maintainability Level, Average Component Dependency (ACD), and Normalised Cumulative Component Dependency (NCCD), which is derived from Cumulative Component Dependency (CCD) and Propagation Cost, are analysed.

Maintainability Level is a percentage that aims to measure the level of maintainability of the application. It is a percentage, so it varies from 0 to 100; the higher, the better. This metric looks at the dependency structure between all the components, taking into consideration the cyclic dependencies and low-level classes with a lot of incoming dependencies, which have a negative influence on the metric [82].

Average Complexity is a weighted average of Cyclomatic Complexity, which is the number of independent linear paths through an application's code. This metric is calculated by the formula:

$$CC = E - N + 2P$$

Where E is the number of edges – connections between nodes in the control flow graph, N is the number of nodes – decision points and statements, and P is the number of connected components – usually 1 for a single program or function.

A higher value means more complexity in the system, having less maintainability since it is harder to understand the code, therefore it takes longer to modify it.

The Average Component Dependency (ACD) is the average number of dependencies of a component (class, package or module). Dividing the Cumulative Component Dependency (CCD), which is the sum of all component dependencies in a system, by the number of modules in the system, we have the ACD. The lower the number, the less coupling, meaning better maintainability. When this value is high, there are more dependencies, making changes riskier.

Normalised Cumulative Component Dependency (NCCD) normalises CCD by comparing it to an ideal reference system of the same size, making it easier to compare different projects, with the following formula:

$$NCCD = \frac{CCD}{N \log_2 N}$$

Where N is the number of components. When the value is approximately 1, the system has a balanced dependency structure; when the value is higher than 1, the system has a higher-than-normal coupling, negatively affecting maintainability, and if the value is lower than 1, the system is extremely well structured, achieving low complexity.

Component Dependency measures coupling. This is directly related to technical debt: the higher the coupling, the harder each change is because it can affect other modules.

Propagation cost indicates the proportion of software files that are directly or indirectly linked to each other. This metric is like the cumulative component dependency metric; however, it indicates the files themselves, not modules. This means developers changing the project must modify more files when the value is higher. This metric also correlates with technical debt and developing efficiency, since the higher the value, the more time it will take to make changes to the project.

Table 5 shows the summary of the metrics being gathered regarding maintainability.

Table 5 – Maintainability metrics summary

Metric	Unit	High Values are	Best Value	Worst Value	Metric Summary
Maintainability Level	% (0 to 100)	Positive	**Highest**	**Lowest**	Indicates the ease of maintaining the software based on dependencies and cyclic structures. Higher is better.
Average Complexity	Number	Negative	**Lowest**	**Highest**	Weighted average of code complexity (cyclomatic complexity). Lower indicates simpler code.
Average Component Dependency (ACD)	Number	Negative	**Lowest**	**Highest**	Average number of dependencies per component; lower reduces coupling and improves maintainability.
Normalised Cumulative Component Dependency (NCCD)	Number	Negative	**Lowest**	**Highest**	Normalised measure of coupling compared to the ideal structure, lower is better.
Propagation Cost	Number	Negative	**Lowest**	**Highest**	Proportion of files impacted by changes; lower indicates easier and quicker modifications.

6.1.3 Energy Efficiency

To gather the metric values about energy efficiency, Energy Consumption, CPU (Control Process Unit) Cycles are collected. Additionally, correlating both these metrics, the energy consumed per CPU cycle and CPU Cycles per Joule are both calculated to give a deeper sense of the energy efficiency of both projects.

Energy consumption is tracked by measuring how much energy the system uses, expressed in joules (J). This metric is important because it reflects the energy efficiency of the application. Kepler is used to gather this data, collecting information from various parts of the system, such as the CPU, DRAM, GPUs, and other host components.

CPU Cycles per Task quantifies the total number of CPU cycles required to complete a specific task. A lower CPU cycle means that fewer computational resources are needed, which typically means a reduction in energy consumption. This is because the amount of energy consumed is related to the resource utilisation, meaning that high CPU usage results in more power drawn

and energy consumed [9], [83]. Kepler does not give the CPU Cycles per Task directly; however, multiplying the CPU Time (in milliseconds) by the CPU frequency gives us the CPU Cycles per Task.

By dividing the energy consumed in joules by the number of CPU cycles per task, we get the energy consumed in joules per CPU Cycle, giving a better metric to compare the energy efficiency between the different architectural designs.

Finally, by switching the formula and dividing the CPU cycles per task by the energy consumed, the result gives the number of CPU Cycles per Joule.

Table 6 shows the summary of the metrics being gathered regarding energy efficiency.

Table 6 - Energy efficiency metrics summary

Metric	Unit	High Values are	Best Value	Worst Value	Metric Summary
Energy Consumption	joules (J)	Negative	**Lowest**	**Highest**	The lower the total energy consumed by the system, the better the energy efficiency.
CPU Cycles per Task	Cycles	Negative	**Lowest**	**Highest**	CPU cycles required per task; fewer cycles imply less resource use and energy consumption.
Energy per CPU Cycle	joules per Cycle	Negative	**Lowest**	**Highest**	The amount of energy consumed per CPU cycle is lower, which indicates better efficiency.
CPU Cycles per Joule	Cycles per Joule	Positive	**Highest**	**Lowest**	Number of CPU cycles completed per unit of energy; higher signifies better efficiency.

6.2 Tests

In this section, the several experiments performed are explained, and the results obtained from the three different software architectures are shared, to analyse how modularisation affects the different quality attributes, concluding whether the modular monolithic application can be a middle ground step before the full migration to microservices.

These results are gathered from a controlled environment to ensure external variables do not affect the results obtained. To achieve this, the applications are containerised through Docker, limiting the resources used to achieve better consistency across all the experiments. For the energy efficiency experiment, Kubernetes was used to deploy several applications and services, since Kepler only works with Kubernetes.

Given its current financial situation, based on the business context referenced in Section 5.1.2.1, deploying all services on a single machine is both practical and financially sensible.

It could be beneficial to attempt to simulate artificial network delay using a service mesh, such as Istio. This would simulate a more realistic distributed system, including instances of numerous availability zones. This setup could help provide important information and be representative of advanced technical planning; however, it was discarded as not being feasible under current fiscal constraints and the possible future growth of the veterinary clinic in the preceding narrative.

All the applications - non-modular monolithic, modular monolithic and each service from the microservices application- are containerised, as mentioned before, with hardware restrictions to perform a transparent experiment.

The hardware and operating system used to evaluate the performance and maintainability are the following:

- Operating System: Windows 11 Pro
- CPU: AMD Ryzen 7 5800X
- Memory: 32GB DDR4 3600Mhz RAM

The same hardware was used to evaluate the Energy Efficiency as a host computer; however, the following differences were applied to the virtual machine used:

- Operating System: Ubuntu 22.02.4 LTS
- CPU: AMD Ryzen 7 5800X, limited to 8 Logical Processors
- Memory: 16GB DDR4 3600Mhz RAM

6.2.1 Performance

To perform performance tests and evaluate how the performance quality attribute behaves across non-modular monolithic, modular monolithic, and microservices, the Grafana K6 tool was used.

The tool used for this study is Grafana K6, since it is easy to set up. By being based on scripting, using JavaScript, the author can leverage their knowledge and implement the load tests quicker than using a tool based on a Graphical User Interface [84]. Although the author already had contact with JMeter, this contact was not extensive and was not a deciding factor in choosing it. To implement load tests quicker, not focusing on the learning of the tool, the Grafana k6 was chosen over the JMeter tool, adding to the other reasons, such as being open-source.

The tests conducted reflect a real workflow from the user, where the user creates an owner, adds two pets and two visits per pet, and then deletes the owner. The metrics values are

collected based on the POST Request (Owner Creation), which affects only one module, and the DELETE Request (Owner Deletion), which affects three modules (check section 5.1.2.2) are collected during this full workflow. Therefore, the metric values reflect not only the endpoint individually but also the potential interactions between steps, such as database consistency, cascading deletes and in-memory caching, providing a comprehensive view of system behaviour under realistic load.

There are two load test scenarios for both requests, one with 10 concurrent users and another with 100 concurrent users. After, a hypothesis test is conducted to verify if the differences in performance are statistically significant.

6.2.1.1 Load Test

As mentioned, the load test was performed using two different scenarios. The first scenario is a 10-simultaneous-user load test, with a ramp-up and ramp-down of 30 seconds, and staying on 10 simultaneous users for 1 minute, with a total of 2 minutes. The second scenario is the same, but with 100 users. Both these scenarios show differences between all three architectures for both requests.

Before the load test for each application, a dummy test was run so that the applications could leverage their cache functionalities.

The values gathered for this subsection are all publicly available on GitHub [76]. These should be compressed into a zip file, since the data results are exported as CSV files.

Table 7, Table 8, Table 9 and Table 10 show the test results for both scenarios, the first two for the 10 concurrent users scenario and the second two for the 100 concurrent users scenario. These tables present the average response time, the minimum response time, the median response time, the maximum response time, the 90^{th} percentile, the 95^{th} percentile and the throughput. All the measured data are in milliseconds, except the throughput, which is in requests per second. The results show that the microservices perform the best, followed by the modular monolithic and the non-modular monolithic.

Between each table, there is a histogram representing the response time values for each application, in each scenario, displayed in Figure 29, Figure 30, Figure 31, Figure 32, Figure 33, Figure 34, Figure 35, Figure 36, Figure 37, Figure 38, Figure 39 and Figure 40.

Table 7 – Load test for post owner request with 10 concurrent users

Architectural Design	Average Response Time (ms)	Min (ms)	Median (ms)	Max (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Non-modular Monolithic	9.79	0.50	1.58	170.66	9.08	84.22	5.99
Modular Monolithic	6.66	0.50	1.07	94.40	3.10	81.76	8.84
Microservices	4.22	0.50	1.89	165.65	3.13	6.26	26.07

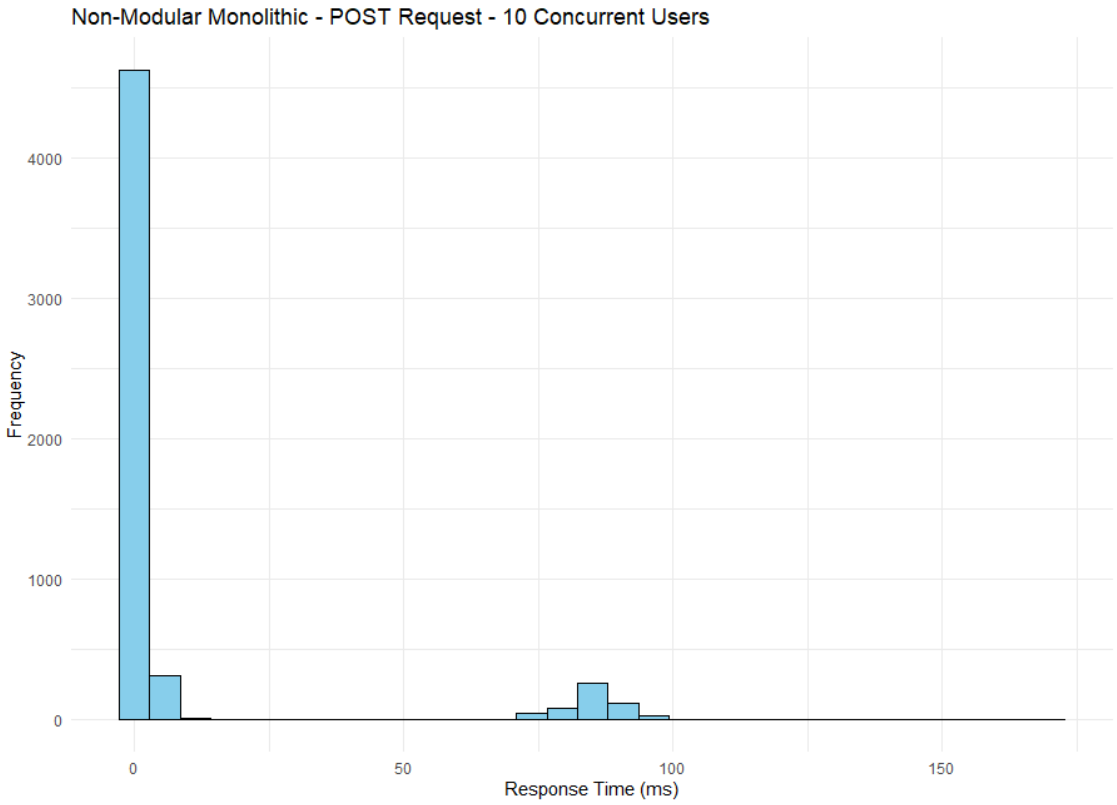


Figure 29 – Histogram for non-modular monolithic POST Request with 10 Concurrent Users

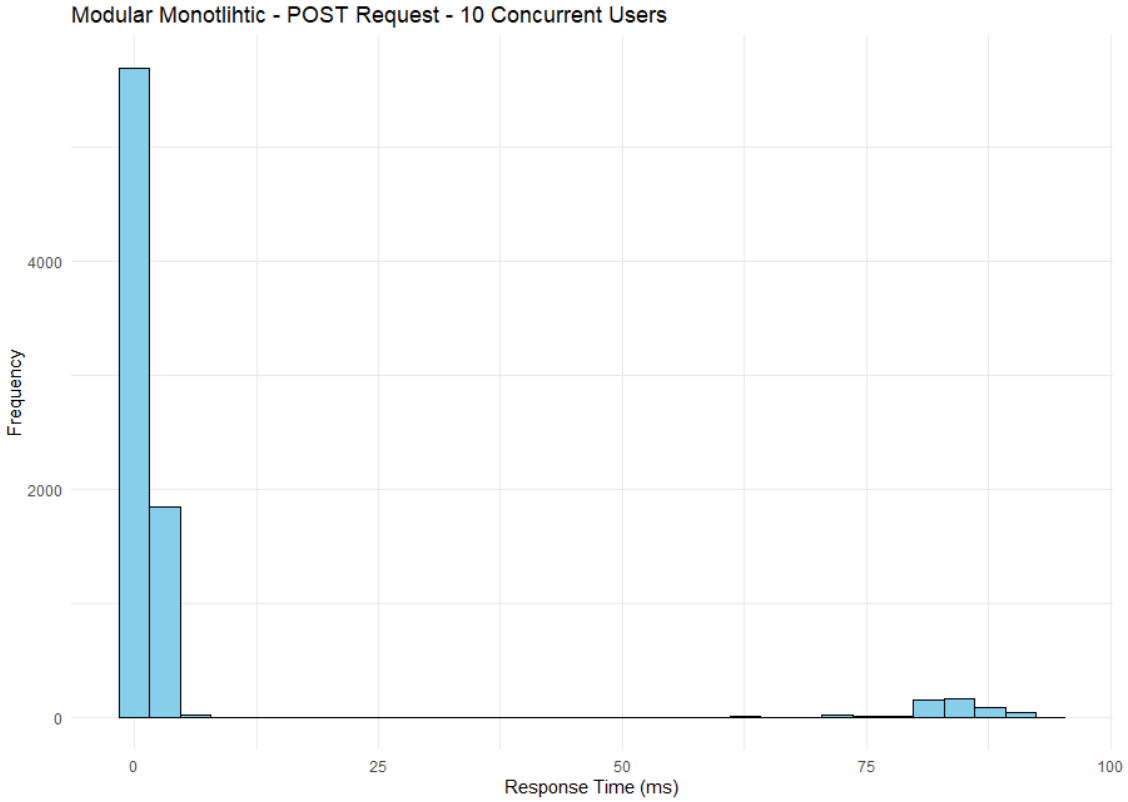


Figure 30 - Histogram for modular monolithic POST Request with 10 Concurrent Users

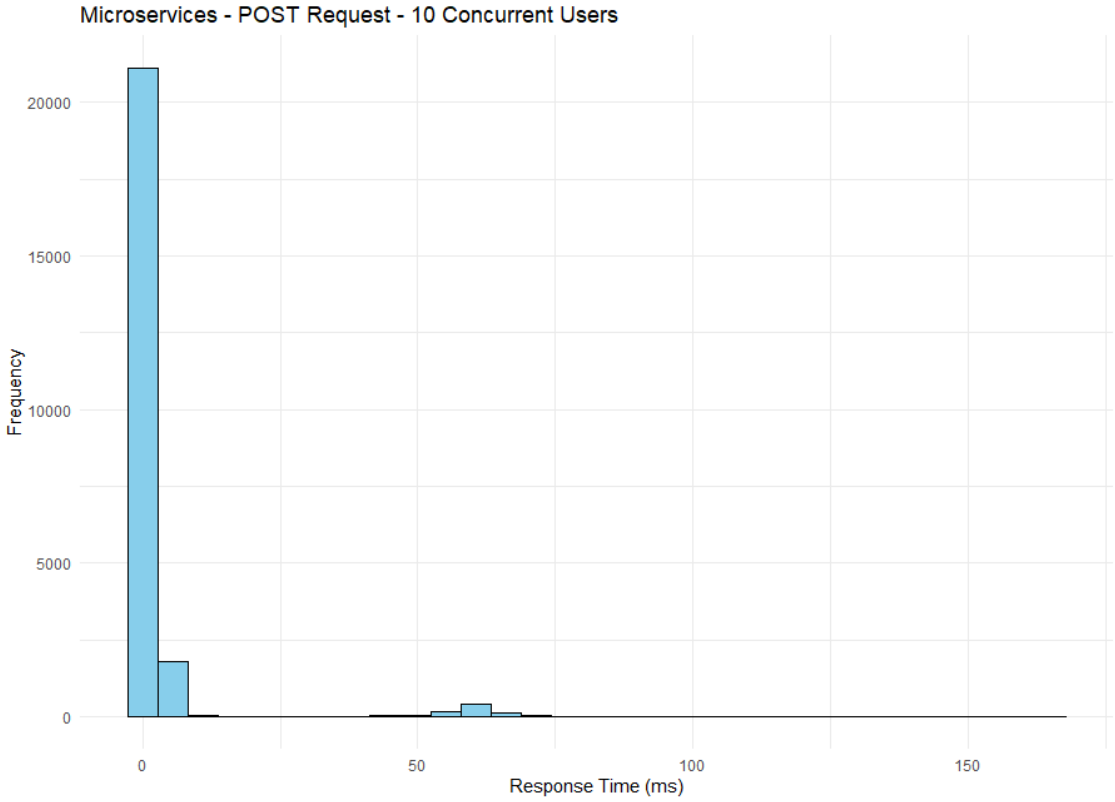


Figure 31 - Histogram for microservices POST Request with 10 Concurrent Users

Table 8 – Load Test for delete owner request with 10 concurrent users

Architectural Design	Average Response Time (ms)	Min (ms)	Median (ms)	Max (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Non-modular Monolithic	12.99	0.50	2.10	171.17	83.03	87.74	5.99
Modular Monolithic	7.99	0.50	1.57	96.36	4.15	82.71	8.84
Microservices	4.18	0.50	1.70	165.68	2.86	6.10	26.07

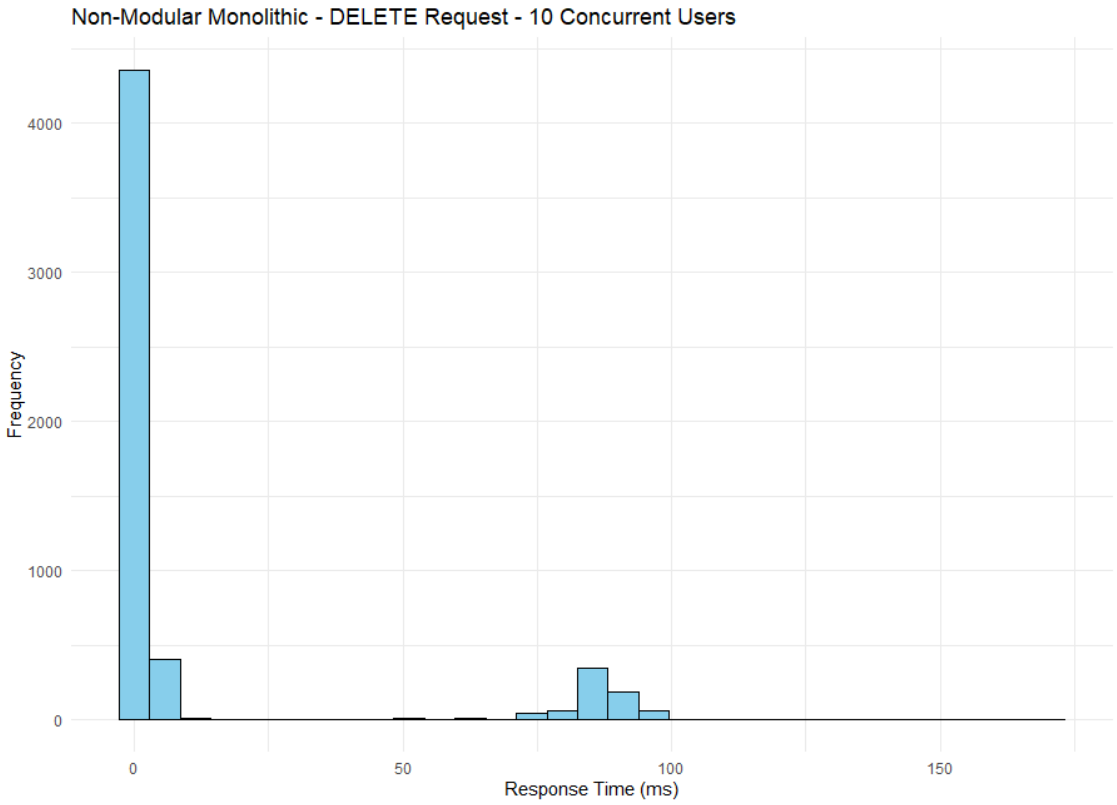


Figure 32 - Histogram for non-modular monolithic DELETE Request with 10 Concurrent Users

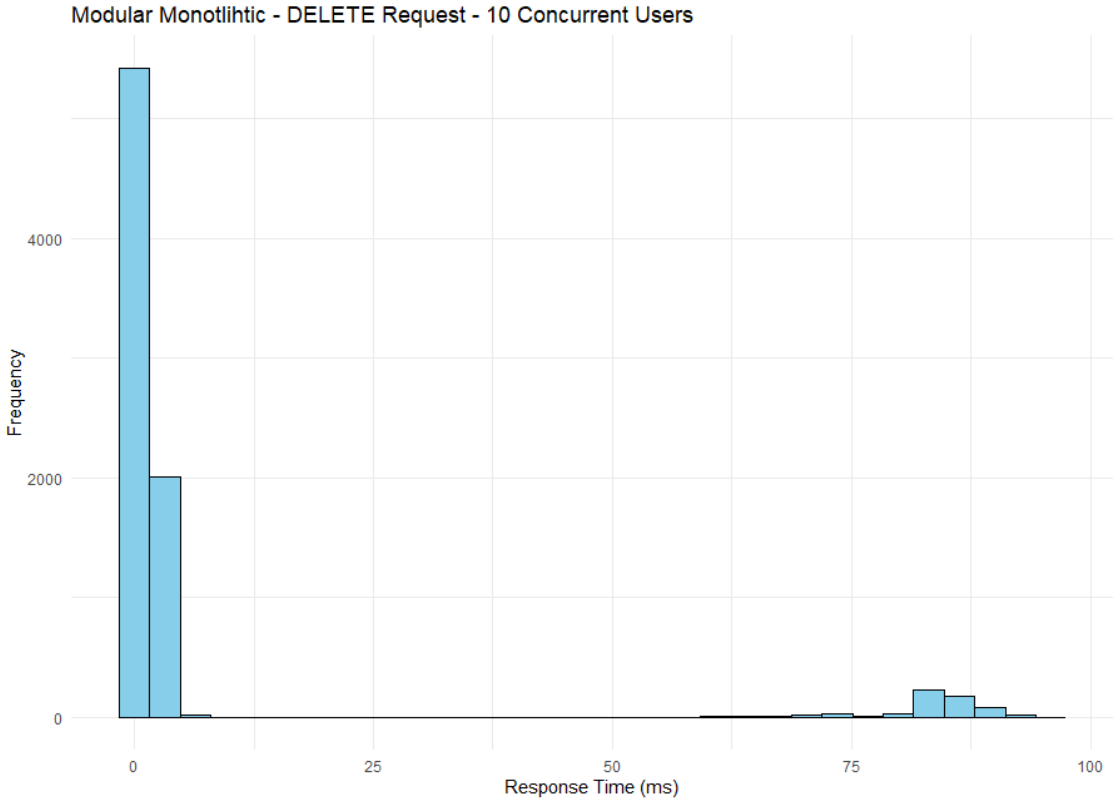


Figure 33 - Histogram for modular monolithic DELETE Request with 10 Concurrent Users

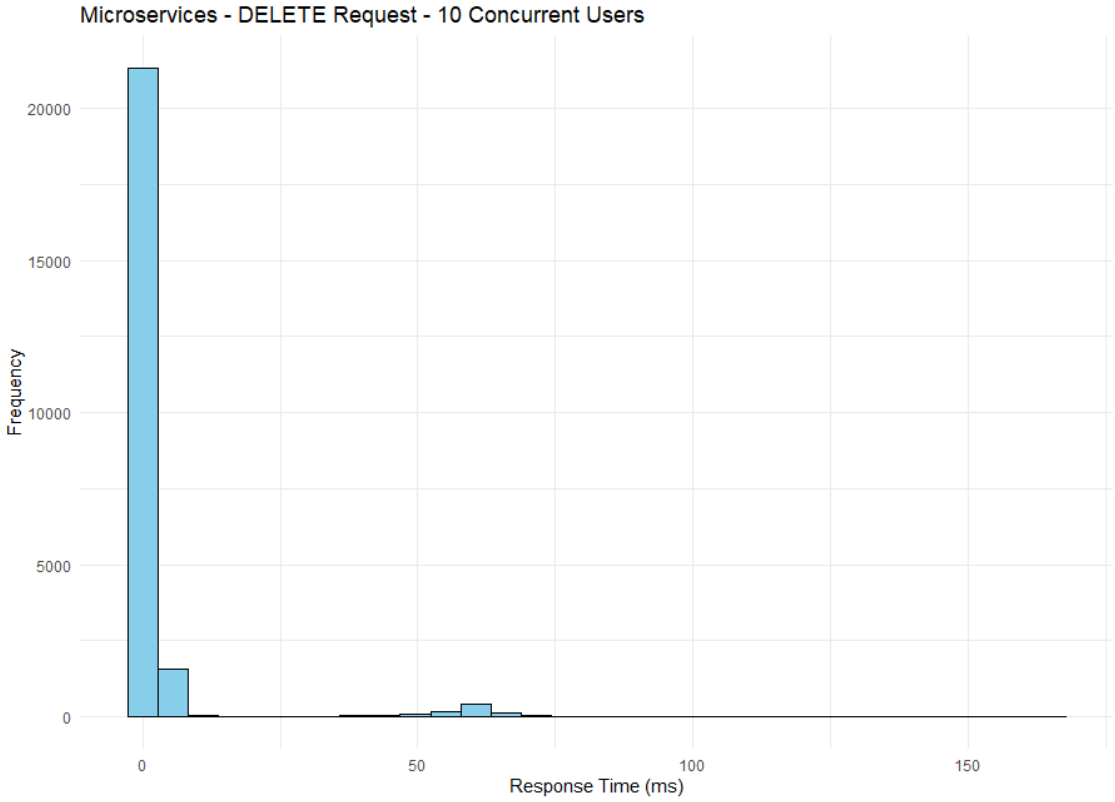


Figure 34 - Histogram for microservices DELETE Request with 10 Concurrent Users

Table 9 - Load test for post owner request with 100 concurrent users

Architectural Design	Average Response Time (ms)	Min (ms)	Median (ms)	Max (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Non-modular Monolithic	64.46	0.50	90.46	584.29	106.44	189	0.92
Modular Monolithic	48.62	0.50	10.97	295.44	99.97	104.84	1.25
Microservices	46.02	0.51	18.57	320.35	94.75	99.14	2.50

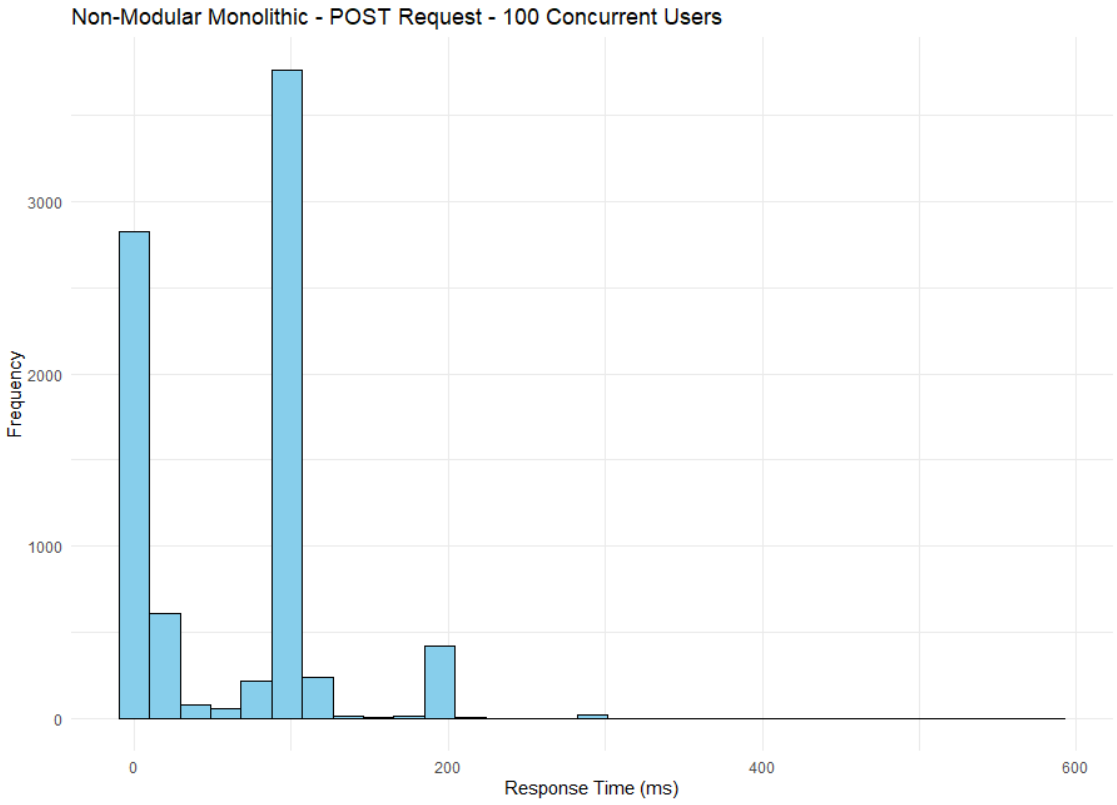


Figure 35 - Histogram for non-modular monolithic POST Request with 100 Concurrent Users

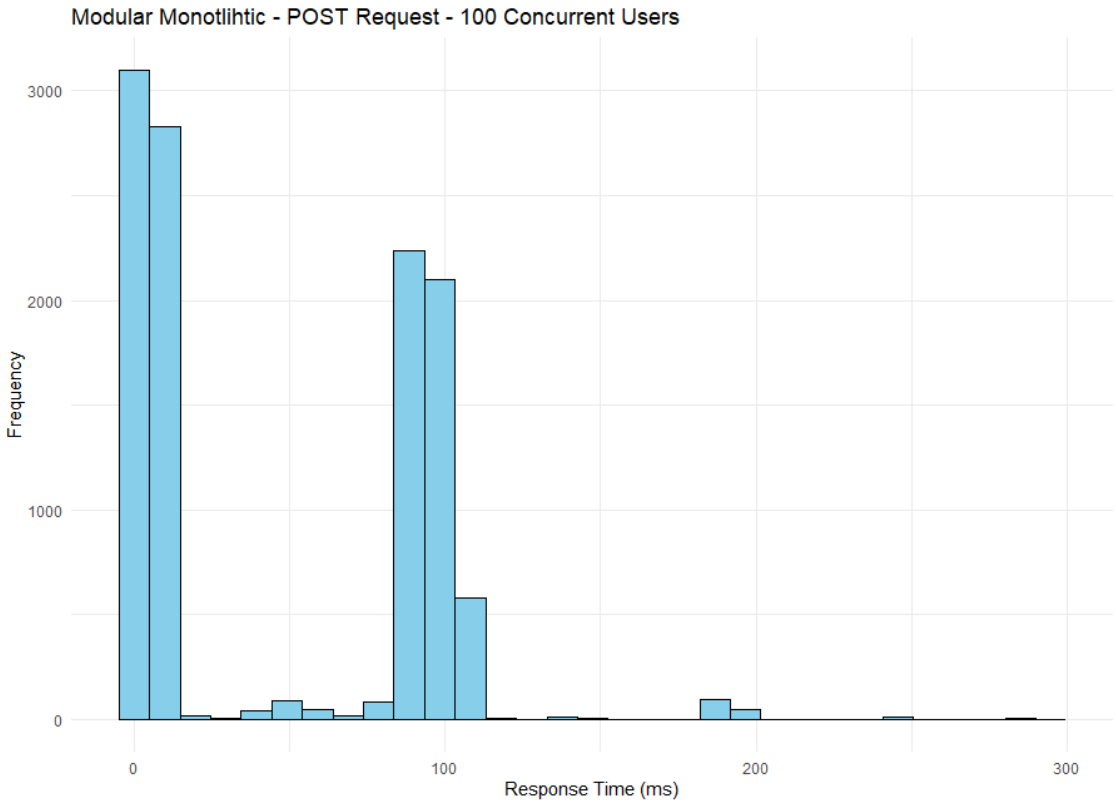


Figure 36 - Histogram for modular monolithic POST Request with 100 Concurrent Users

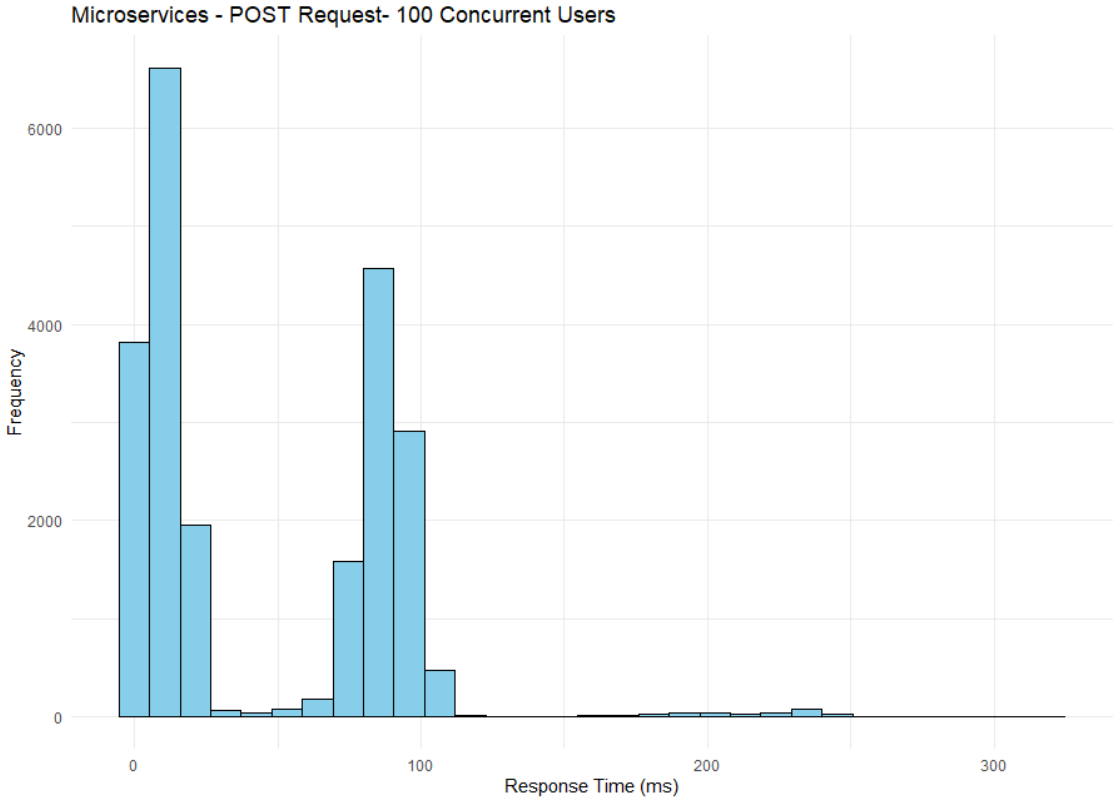


Figure 37 - Histogram for microservices POST Request with 100 Concurrent Users

Table 10 – Load Test for delete owner with 100 concurrent users

Architectural Design	Average Response Time (ms)	Min (ms)	Median (ms)	Max (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Non-modular Monolithic	76.25	0.52	92.97	599.1	187.09	193.65	0.92
Modular Monolithic	51.41	0.50	14.17	325.54	101.30	105.89	1.25
Microservices	40.38	0.50	14.89	248.96	92.64	97.45	2.50

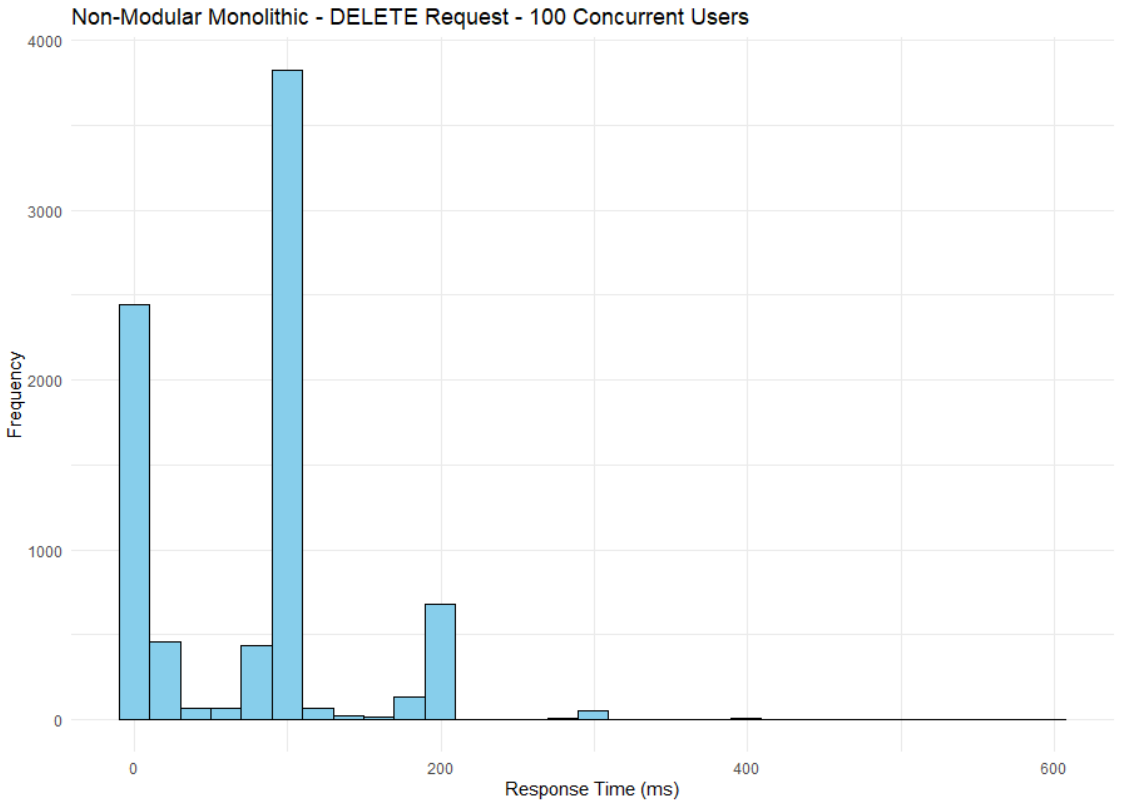


Figure 38 - Histogram for non-modular monolithic DELETE Request with 100 Concurrent Users

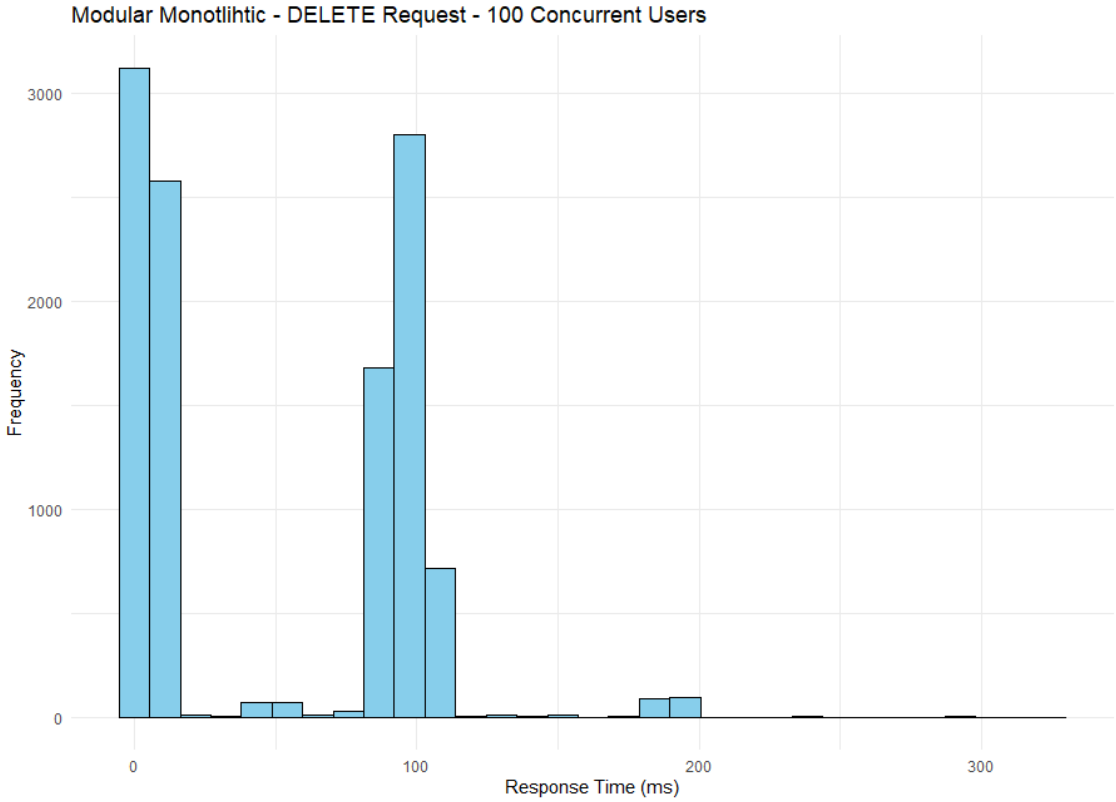


Figure 39 - Histogram for modular monolithic DELETE Request with 100 Concurrent Users

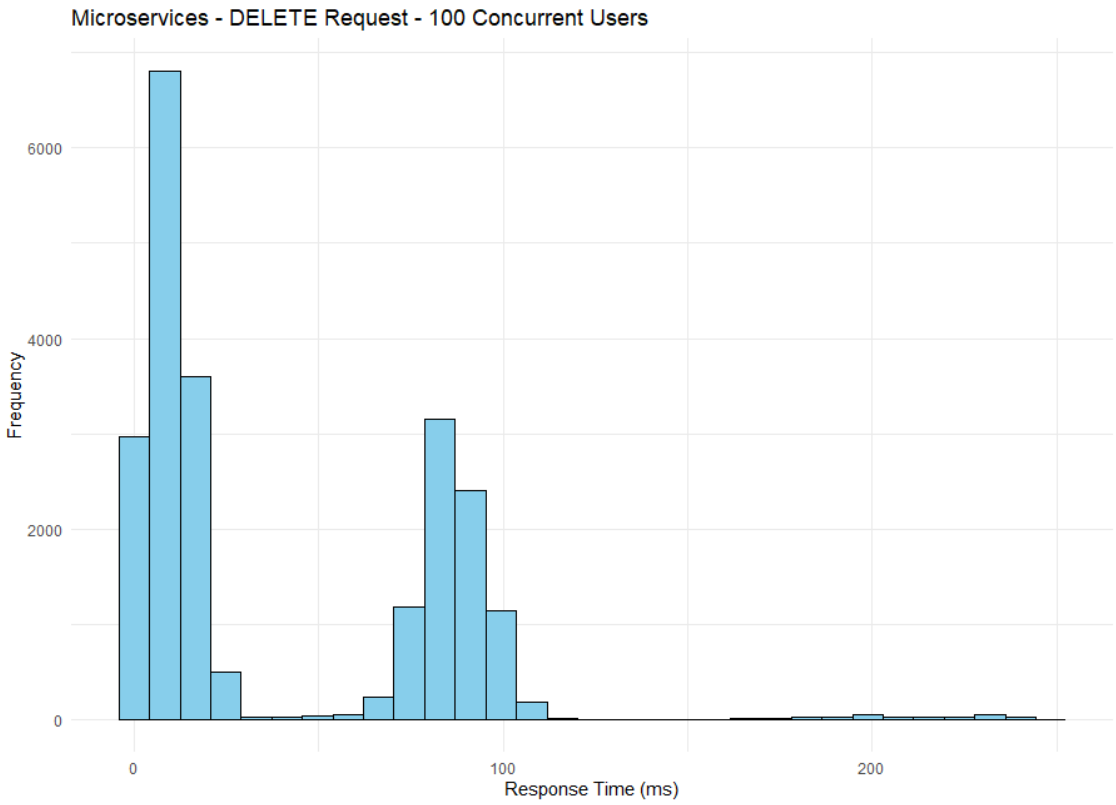


Figure 40 - Histogram for microservices DELETE Request with 100 Concurrent Users

Across all scenarios and request types, the error rate was consistently 0%, with the sole exception of the modular monolithic architecture. In this particular setup, the error rate was extremely low: 0.0000072% for 10 concurrent users and 0.0000110% for 100 concurrent users. These minor errors occur because the modular monolithic relies heavily on asynchronous calls. While this architecture typically avoids database conflicts due to the use of a single database, occasional simultaneous requests to the same database rows can cause rare conflicts, resulting in these minor errors.

For the POST request scenario with up to 10 concurrent users, the non-modular monolithic architecture recorded an average response time of 9.79 ms, with a minimum of 0.50 ms, a median of 1.58 ms, and a maximum of 170.66 ms. The 90^{th} and 95^{th} percentiles were 9.08 ms and 84.22 ms, respectively, with a throughput of 5.99 requests per second.

In comparison, the modular monolithic setup performed better, with an average response time of 6.66 ms, minimum of 0.50 ms, median of 1.07 ms, and maximum of 94.4 ms. Its 90^{th} percentile was notably lower at 3.10 ms, with the 95^{th} percentile at 81.76 ms, and a higher throughput of 8.84 requests per second.

The microservices architecture performed best overall, achieving an average response time of just 4.22 ms, the same minimum response time of 0.50 ms, a median of 1.89 ms, and a maximum of 165.65 ms. Crucially, its 90^{th} percentile was 3.13 ms, and the 95^{th} percentile was significantly lower at 6.26 ms, resulting in a high throughput of 26.07 requests per second.

While the median response time for microservices appears higher at first glance compared to the monolithic architectures, it is important to interpret this in the context of the higher volume of requests handled. The microservices system demonstrates notably better consistency and predictability, as evidenced by its significantly lower 95^{th} percentile and substantially higher throughput. Thus, the slightly higher median value does not reflect inferior performance but rather highlights its robust performance under greater load and concurrency.

For DELETE requests under the same conditions (10 concurrent users), the non-modular monolithic setup again performed the poorest, with an average response time of 12.99 ms and a throughput of 5.99 requests per second. The modular monolithic performed better, averaging 7.99 ms with a throughput of 8.84 requests per second. The microservices setup was superior, achieving an average response time of 4.18 ms and throughput of 26.07 requests per second. While the microservices setup had a slightly higher median (1.70 ms) compared to the modular monolithic (1.57 ms), its lower 90^{th} percentile (2.86 ms) and significantly lower 95^{th} percentile (6.10 ms) illustrate its greater consistency under load.

In scenarios involving 100 concurrent users, performance differences became more pronounced. For POST requests, the non-modular monolithic setup had the weakest performance with an average response time of 64.46 ms and throughput of only 0.92 requests per second. The modular monolithic improved upon this, with an average response time of 48.62 ms and throughput of 1.25 requests per second. The microservices setup slightly outperformed the modular monolithic, averaging 46.02 ms and maintaining higher throughput

at 2.50 requests per second. Although the median response time was higher for microservices (18.57 ms compared to 10.97 ms), the better 90th and 95th percentiles and higher throughput demonstrate greater efficiency and scalability under higher loads. However, as these values are very approximate, the hypothesis tests conducted further in the document will conclude if there is a statistically significant difference between these two architectures in this test scenario.

For DELETE requests at 100 concurrent users, microservices again led with an average response time of 40.38 ms and throughput of 2.50 requests per second. The modular monolithic followed with 51.41 ms and 1.25 requests per second, and the non-modular monolithic performed the poorest with 76.25 ms and a throughput of 0.92 requests per second. Although the microservices' median (14.89 ms) was slightly higher than the modular monolithic's (14.17 ms), its better 90th percentile (92.64 ms) and significantly improved 95th percentile (97.45 ms) reinforce its stable and predictable performance even under heavy load.

To confirm these observations, hypothesis testing will later establish whether the performance differences between these architectures are statistically significant.

6.2.1.2 Hypothesis Test

To statistically validate the differences observed in the load tests between the three architectures' applications, the R programming language with an R script was used to perform hypothesis tests. By performing hypothesis tests, it is possible to statistically affirm or negate the performance difference between all the types of architecture designs.

The first step is to check the normality of the dataset. Given that the dataset of the requests is bigger than 30, the *Lilliefors* test was performed, instead of the *Shapiro-Wilk*, which is done for samples smaller than 30. After the normality has been checked, the symmetry of the datasets was checked, which informs whether parametric or non-parametric hypothesis tests are conducted. For the symmetrically distributed data and followed by a normal distribution, a parametric test like the *t-test* is done; on the contrary, a non-parametric test is performed, such as the *Wilcoxon signed-rank* test.

The results for these hypothesis tests in this subsection are publicly available on GitHub [76].

To compare the results of the load tests, a total of 3 hypothesis tests were conducted, each one of them for both 10 and 100 concurrent users, between the POST Request and the DELETE Request:

- Non-Modular Monolithic vs. Modular Monolithic
- Non-Modular Monolithic vs. Microservices
- Modular Monolithic vs. Microservices

The R script used to run the hypothesis is available in Appendix C. This script shows the menu for the script to be run, and then it prints all the results. This script and all the results are available on GitHub [76].

The results obtained from the scenario with 10 concurrent users for the POST Request available on Appendix D, with 10 concurrent users for the DELETE Request available on Appendix E and with 100 concurrent users for the DELETE Request available on Appendix G all indicate that the hypothesis tests consistently rejected the null hypothesis (H0) - the non-existence of statistically significant differences between the data sets. Instead, the tests support the alternative hypothesis (H1), meaning that the data analysis offers significant statistical differences.

On the contrary, the results obtained from the scenario with 100 concurrent users for the POST Request available on Appendix F, between the modular and the microservices architecture, indicate that the hypothesis tests do not reject the null hypothesis (H0), suggesting no statistically significant difference between these two architectures in this scenario.

Given these results, it is possible to statistically assume that the aforementioned assumptions, which say that the microservices perform the best, followed by the modular monolithic and then by the non-modular monolithic, hold in load test scenarios, except for the scenario which only affects one module in every architecture, which is not guaranteed.

Since this performance test collects metric values from the DELETE and POST Requests from a workflow point of view, and not individually, at a high load, the limiting factor becomes the database, and since the database is the same in both applications, even though the microservices has one database for each service, this POST Request is just testing one service/module, which levels the performance between the modular and microservices. But when there are more modules affected, the microservices application still outperforms the modular monolithic application. For the low load, the microservices application holds better results for its app-level isolation. Also, the throughput of the microservices application is higher in every test, meaning a bigger number of requests per time frame, showing once more the power of this architectural style on the performance quality attribute.

Moreover, modularity positively affects performance, and there is a statistically proven difference between these three architectural designs. Even though the modular monolithic application and microservices application perform the same in a single scenario, they all outperform the non-modular monolithic application in every case. Therefore, the modular monolithic design is a good middle-step migration design between the non-modular monolithic and microservices when affecting the performance quality attribute.

6.2.2 Maintainability

To assess maintainability and measure the metrics referenced in the GQM, the tool Sonargraph is used, which is referenced in the subsection 3.1.1.1. This tool is helpful since all the metric values about maintainability needed are available when the application is loaded into the tool.

Each application from the different architecture types was loaded into Sonargraph, which gives the metrics.

As referenced before, the metrics analysed are Average Complexity, Maintainability Level, Average Component Dependency (ACD), and Normalised Cumulative Component Dependency (NCCD) for each architecture type.

All the metric values gathered in this subsection are publicly available on GitHub [76].

The results obtained are shown in Table 11.

Table 11 – Maintainability Metrics Results

Architectural Design	Average Complexity	Maintainability Level	ACD	NCCD	Propagation Cost
Non-modular Monolithic	2.62	94.61	4.92	1.28	20.49
Modular Monolithic	2.34	94.03	3.97	0.94	12.4
Microservices	1.17	95.93	2.14	0.49	5.77

After gathering the metrics values from Sonargraph from all the applications, the trend of the microservices application having better values, followed by the modular monolithic and then by the non-modular monolithic, continues.

The microservices achieve better values in all the metrics represented above; the modular monolithic performs slightly worse, by 0.58, in Maintainability Level.

In terms of average complexity, the microservices stand itself from both monolithic applications, having a value of 1.17 compared to 2.62 and 2.34. Even though the modular monolithic has a worse value than microservices, it still has a better value than the non-modular monolithic, showing that this architecture has benefits, decreasing the complexity.

As said before, in terms of maintainability level, all three applications are close, with the microservices being the best one, second the non-modular monolithic and then the modular monolithic. The difference is minimal, but since the modular monolithic has more components than the non-modular monolithic, for example, the classes that manage domain events, this value can be slightly lower.

The average component dependency follows the pattern marked by average complexity, with the non-modular monolithic having the worst value, with 4.92, meaning its components have the most dependencies, followed by the modular monolithic, with 3.97, meaning when applying modular technics in a monolithic, the dependencies between components decrease, and finally, the microservices with the lowest value, 2.14, since the application is composed by a group of independent services.

Analysing Normalised Cumulative Component Dependency, which is a normalised metric, removing the size of the application from the equation, the non-modular monolithic has the worst value, 1.28, followed by the modular monolithic, 0.94, and finally, the microservices with 0.49.

Last, the propagation cost also follows the trend from before, the non-modular monolithic has the worst value, 20.49, the modular monolithic comes closer to the microservices application, with 12.4, having a big beneficial decrease on this metric value, meaning that when changing some component, their propagation will not cost as much as in the non-modular monolithic. Finally, the microservices application has a value of 5.77, being the best one in this metric.

Concluding, even though the microservices have better results in terms of maintainability, the modular monolithic approach continues to benefit from the modularity implemented in the application, with slightly better results than the non-modular monolithic approach. Thus, the modular monolithic is a competent solution to be a middle step design between the monolithic and microservices migration.

6.2.3 Energy Efficiency

Finally, the energy efficiency is measured. The metrics – energy consumed in joules, number of CPU Cycles, the energy consumed per CPU cycle and CPU Cycles executed per Joule are used the metrics obtained to evaluate this quality attribute.

The tool used was Kepler, since it is free of charge, open-source and well-documented. Since Kepler uses Kubernetes clusters to measure the data, there is a need to deploy the three applications to Kubernetes, as mentioned before. Additionally, Kepler only works with Linux systems, meaning the metrics gathering was performed using a Virtual Machine.

Even though Kepler did not give the number of CPU Cycles directly as a metric, the CPU Time in Milliseconds is available. Since the CPU used has a maximum of 3 800 MHz clock frequency, which is a value that varies in real-time, CPU Cycles are calculated by obtaining the CPU Time in seconds and multiplying it by the clock frequency in Hertz (3 800 000 000).

To obtain the Energy Consumed per CPU Cycle, the energy consumed is divided by the CPU Cycles count. To obtain the CPU Cycles performed by Joule, the CPU Cycles count is divided by the Energy consumed.

The procedure had the following workflow: first, set up Kepler; after, deploy the application into Kubernetes inside the Virtual Machine; followed by forwarding the ports so that it is possible to access the application, then, both load tests, with 10 and 100 concurrent users, developed for the performance analysis were run, so that Kepler registers the data needed.

All the metric values gathered in this subsection are publicly available on GitHub [76].

Table 12 shows the data for the 10 concurrent users scenario and Table 13 shows the data for the 100 concurrent users scenario.

Table 12 – Energy Efficiency Metrics Load Test with 10 Concurrent Users

Architectural Design	Energy Consumed (J)	CPU cycles	Energy Consumed (J) per CPU cycle	CPU cycles per joule
Non-modular Monolithic	979.75	4.15E+11	2.36E-09	423 467 235
Modular Monolithic	1 044.87	4.20E+11	2.49E-09	401 816 106
Microservices	4 872.41	1.48E+12	3.29E-09	303 697 305

Table 13 - Energy Efficiency Metrics Load Test with 100 Concurrent Users

Architectural Design	Energy Consumed (J)	CPU Cycles	Energy Consumed (J) per CPU cycle	CPU cycles per joule
Non-modular Monolithic	808.82	3.19E+11	2.54E-09	393 910 890
Modular Monolithic	945.66	3.72E+11	2.54E-09	393 671 430
Microservices	5 370.78	1.59E+12	3.37E-09	296 865 128

These two tables show the difference between the data gathered immediately before and immediately after running the tests.

In case of the microservices, it includes the sum from all services needed, these include – the Configuration Service, the Discovery Service, the Kafka Service, the Owners Service, the Pets Service, the Visits Service and the Vets Service. The data from the energy consumed is based only on load, not including the energy consumed when the service is idle.

The non-modular monolithic is the application that consumes the least energy, 979.75 J for the 10 Concurrent Users scenario and 808.82 J for the 100 concurrent users scenario. Additionally, it is also the application that has fewer CPU Cycles, with 4.15E+11 for the 10 concurrent users scenario and 3.19E+11 for the 100 concurrent users scenario. This means that the Energy Consumed per CPU Cycle for the first scenario is 2.36E-09 joules, and for the second scenario is 2.54E-09 joules. The CPU Cycles per Joule for the first scenario is 423 467 235, and for the second scenario is 393 910 890.

The modular monolithic application consumed slightly more, with 1 044.87 J for the 10 concurrent users scenario and 945.65 J for the 100 concurrent users scenario. For the CPU Cycles, when running the load test with 10 concurrent users, the application counted 4.20E+11 CPU Cycles for the 10 concurrent users scenario and 3.72E+11 for the 100 concurrent users scenario. The energy consumed per CPU Cycle is 2.49E-09 joules for the first scenario and 2.54E-09 joules for the second scenario, and the CPU Cycles per joule are 401 816 106 for the scenario with 10 Concurrent Users and 393 671 430 for the scenario with 100 Concurrent Users.

Finally, consuming the most energy based on the data, the microservices application consumed 4 872.41 joules for the scenario with 10 concurrent users and 5 370.78 joules for the scenario

with 100 concurrent users. The CPU Cycles Count for the first and second scenarios are 1.48E+12 and 1.59E+12. This means that the Energy Consumed per CPU Cycle for the first scenario is 3.29E-09 joules and 3.37E-09 joules for the second scenario. Lastly, the CPU Cycles count per joule for the microservices application, for the 10 concurrent users Scenario and 100 concurrent users Scenario, respectively, are 303 697 305 and 296 865 128.

The microservices application consumes more energy than both monolithic applications by a big margin. The fact that each service is an independent application contributes to this. Additionally, the microservices application has an overhead of serialisation/deserialization based on the request between services, which needs more CPU usage, meaning a higher consumption of energy [12], [85].

Another topic that can lead to more energy consumption is the cache strategy. The monolithic applications are only one encapsulated application, meaning that data in memory can be cached automatically by the application itself. Compared to the microservices application, this strategy only applies to the service itself, or there is a need to implement a different strategy [86]. This is also why the higher the count of concurrent users, the monolithic the applications become (fewer CPU Cycles per Joule) than the microservices, which tends to go the opposite way.

Between the monolithic applications, the modular application has slightly less energy efficiency than the non-modular monolithic application, which tends to be less noticeable the more concurrent users. Since the modular monolithic needs more classes to manage the domain events, the application tends to need more energy and have less energy efficiency.

In performance and maintainability, the microservices application has better results, followed by the modular monolithic and then by the non-modular monolithic. In contrast, the trend changes for energy efficiency, the non-modular monolithic has the best energy efficiency, followed by a small margin by the modular monolithic application and then by a big margin by the microservices application.

Energy efficiency can be a factor that gives the modular monolithic design an advantage over the non-modular monolithic design and microservices. Although benefits in performance and maintainability are not as high as those of microservices, this architectural design has only a slight decrease in energy efficiency compared to the non-modular monolithic, and microservices have a high increase, giving one more reason to adopt this architecture.

6.3 Summary

Concluding this section, after defining the methodology using the Goal-Question-Metric approach - specifying the goal, refining the questions and selecting appropriate metrics - the metric values were gathered to answer the defined questions and ultimately to achieve the established goal.

Regarding performance, the results from the load tests and the hypothesis tests align with the literature reviewed, confirming better performance in applications with increased modularity. Only in one case did the microservices application not outperform the other two architectures, in high load, with a single module being affected. This happens, as mentioned before, because the limiting factor at high load is the database, which, in comparison, these two architectures, with only one module being affected, is the same. Even though the microservices did not outperform the modular monolithic, bear in mind that the modular monolithic did not also outperform the microservices. Hypothesis tests conclude that there was no significant statistical difference between these two architectures.

However, the microservices still had higher throughput, meaning that more requests were performed in comparison to the modular monolithic, which can impact these values, by receiving more requests per second, the system has a higher load to deal with, concluding that the microservices application can handle more requests with the same response time as the modular monolithic.

The median is also worse in the microservices application in several scenarios compared with the modular monolithic applications, but, once again, this can be affected by the size of the dataset. The microservices application has a higher throughput, meaning a higher request count, and the microservices application still outperforms the modular monolithic in these cases because it has a better 90^{th} percentile and a better 95^{th} percentile.

Nevertheless, the results from the metric values gathered still confirm that the modularity has positive impacts on the performance of an application, and the modular monolithic performs at a higher level than the non-modular monolithic architecture and proves to be a serious competitor for the microservices architecture.

The literature mentions the potential for modularity to introduce performance overheads; this experiment showed no such negative impact. Instead, higher modularity resulted in better performance, suggesting that the implemented modular structures were efficient enough to remove any overhead.

For maintainability, the gathered metric values also confirm what the literature mentions, that modularity positively impacts maintainability. By refactoring the application into clearly defined modules, maintainability improved noticeably. Modules with fewer dependencies reduce the overall complexity, leading to easier and less risky code changes.

Finally, regarding energy efficiency, the collected metric values somewhat support what the literature mentions. Increased modularity correlates with decreased energy efficiency due to higher energy consumption, removing code smells, for example. This occurs because modular systems often involve additional computational overhead from managing interactions between multiple independent modules, leading to higher energy usage. However, it is also mentioned that when an application has better modularity, this could lead to a higher energy efficiency, which did not happen.

In summary, this experiment validated the expected benefits and trade-offs of modularity presented in the literature, affirming performance and maintainability improvements while acknowledging the energy efficiency costs associated with higher modularisation.

7 Conclusion

This chapter highlights the main achievements and contributions of this work, beginning with section 7.1. Section 7.2 discusses the challenges encountered throughout the research process, while Section 7.3 addresses potential threats to validity. In Section 7.4, possible directions for future work are explored. Finally, Section 7.5 presents the concluding remarks. The chapter also provides a concise overview of the key findings presented throughout the dissertation.

7.1 Achievements and Contributions

This study successfully achieved its primary objectives, which were to demonstrate the impact of modularity in an application on the following software quality attributes: performance, maintainability and energy efficiency and conclude whether the modular monolithic approach can be considered a middle ground step between the non-modular monolithic and microservices.

By comparing three different architectures, non-modular monolithic, modular monolithic and microservices, which gradually increase the modularity level of an application in a controlled experience and doing and analysing different metrics, the study achieved an understanding of modularity trade-offs.

One of the significant achievements of this study includes the adaptation of the microservices application to the same class structure and functionalities that the non-modular and modular applications already had available. By implementing Domain Driven Design, with the use of Kafka, it was possible to make a fair comparison between all three architectural strategies, allowing for transparency in the case study.

The load tests developed using Grafana K6 contributed quality data on the performance between non-modular, modular monolithic architecture and microservices architecture. This data helped to recognise the advantages of performance of modularity in an application, achieving better results on microservices, followed by the modular monolithic and then by the non-modular monolithic and addressing the benefits of the modular monolithic. To validate the comparison, hypothesis tests were conducted, confirming the differences were statistically significant between the three architectures.

The maintainability analysis followed the trend mentioned before. By using Sonargraph as a tool to gather metrics about this software quality attribute, it was possible to analyse and understand the given values, confirming once again the advantages of modularity, and demonstrating benefits from the modular monolithic approach meaning that modularity can increase the maintainability of an application.

Finally, the energy efficiency was also measured and analysed; by using Kepler as the tool to gather the needed metrics, it was possible to conclude how the energy is affected after increasing modularity. However, in this software quality attribute, the trend is contradicted, given that the microservices application has less energy efficiency than the monolithic applications. Between the two monolithic applications, the modular architecture also had a lower energy efficiency than the non-modular architecture, however, this difference is not so abrupt, meaning that the modular monolithic can take advantage of the increase in performance and maintainability and stay close to the energy efficiency of the non-modular monolithic.

Moreover, the results obtained claim the benefits of adopting modularity strategies in the performance and maintainability areas. However, in terms of energy efficiency, this tends to get worse, meaning that when choosing an architectural strategy for the application to be developed or to be migrated, it is needed to analyse all the trade-offs.

7.2 Difficulties

In this study, various difficulties and challenges were encountered. In the migration process, the need to achieve the same functionalities between the three applications was a challenge, since in the microservice application, these were not developed. The need to understand and learn new software tools, such as Kafka, to implement the Event Driven Design was a challenge that consumed time.

During the evaluation process, the need to learn about the tools to be used was also time-consuming and a challenge for the writer, since for some of them, it was the first time interacting with them, including the development of Grafana K6 scripts for load testing and Kepler. The need to install, configure a virtual machine using Linux and to understand how to deploy the three applications to Kubernetes, so Kepler could obtain truthful data, was also very time-consuming and difficult.

These difficulties highlight the complexity involved in this study, including the knowledge to master three different architectural strategies and the need to understand and know a set of tools and software, including Docker, Grafana K6, Kepler, Kubernetes and Virtual Machine configuration.

7.3 Threats to Validity

In this study, several factors can affect the results' validity. One important factor is the application used to compare the three architectural strategies. To ensure a fair comparison, the same application in the three architectural strategies already being developed was an important factor since the migration process from scratch was not the focus of this study. However, the microservices application lacked functionalities, and the component/service

arrangement was not the same as that of monolithic applications. This means that there is not full guarantee that the three applications are the same, since they differ from architecture standards.

Adding to this, only one project was analysed, meaning that the size, type, and component structure are the same, making it impossible to know if modularity impacts can change based on the application context.

One potential threat to internal validity lies in the distribution of computational resources across the different architectures, particularly in the case of the microservices application. Since microservices are composed of multiple independently deployed services, each running in its own container, the total amount of allocated resources across the system is inherently higher than in the monolithic applications configurations, which run as a single containerised application. While per-service limits were matched to the container of the monolithic applications to ensure parity at the unit-of-work level, the aggregate resource availability in the microservices deployment may still influence performance outcomes.

When comparing energy efficiency values, it's important to note that while the CPU's advertised maximum clock speed is 3 800 MHz, the actual frequency often varies based on how the hardware reacts to the workload. This fluctuation can affect the accuracy of the energy efficiency measurements. On top of that, the tool used to gather this data, Kepler, relies on estimates rather than exact readings, which adds another layer of uncertainty to the results. These points should be kept in mind when evaluating the findings.

The choice of tools used to measure the metrics can also be a threat to validity, meaning there is no confirmation that the tools are giving the correct information. To combat this, the tools used were also chosen for their quality and have already proven to be reliable.

It is also important to recognise that the author is not the most experienced in all these tools and software used. Although a significant effort was made to minimise the impact of the learning curve, the results could differ if the author had already mastered the tools used.

7.4 Future Work

There are several promising directions for future research. As a continuation of this study, tests could be done in more request types, such as GET. Additionally, these requests could be compared with the number of modules, for example, compare how the values change when a request affects only 1 module, or more. Another logical step is to analyse how modularity can affect several other software quality attributes, beyond performance, maintainability and energy efficiency, such as scalability, security, reliability, and testability, providing a comprehensive understanding of modularity's broader impact.

Also, as referenced before, conducting another experiment with a service mesh could be beneficial, simulating a realistic distributed environment, including scenarios with multiple availability zones.

It would also be interesting to analyse more applications of different sizes, complexities and domains, to be able to conclude if modularity influences in this way independently of the size and type of application or if the size and type can also be influencers for how modularity influences other software quality attributes.

Another area to study is how modularity can influence the application software life cycle by following it over time, with the intention to demonstrate the long-term effect of modularity on the cost of maintenance, flexibility, and robustness. Other trade-offs could also be analysed, such as cost-benefit, to conclude when and how a migration between architectures would make sense.

Finally, the work done in this project is publicly available on GitHub [76] so the community can leverage this data in future studies about how modularity impacts software quality attributes.

7.5 Final Considerations

This dissertation represents an important milestone achieved by the author, marking the conclusion of a full study cycle in the Software Engineering field. The journey was both challenging and rewarding, since it is in a field of interest for the author, who aspires to influence researchers, students and developers to immerse themselves in this field.

This project was a substantial challenge in the author's life. The need to overcome challenges directly connected to the project itself, such as learning new tools, strategies and others, was a central point, but also on a personal, mental and emotional level, since the results and success of the project directly affect the author's success career and life, given the time consumption and the need to achieve such an important milestone.

Finally, the author grew in personal, emotional, mental and professional aspects because of the process of developing this dissertation, acquiring important knowledge about the software engineering field, metric analysis and modular applications development. Finally, all the mental strength required to finish this project has been hard to achieve, but it has been accomplished, making the author grow as a valuable professional person.

8 Bibliographic References

[1] Sam. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st ed. O'Reilly Media, 2015.

[2] N. C. Mendonca, C. Box, C. Manolache, and L. Ryan, 'The Monolith Strikes Back: Why Istio Migrated from Microservices to a Monolithic Architecture', *IEEE Softw*, vol. 38, no. 5, pp. 17–22, Sep. 2021, doi: 10.1109/MS.2021.3080335.

[3] L. De Lauretis, 'From Monolithic Architecture to Microservices Architecture', doi: 10.1109/ISSREW.2019.00050.

[4] R. Su, X. Li, and D. Taibi, 'From Microservice to Monolith: A Multivocal Literature Review', pp. 10–12, 2023, doi: 10.3390/electronics13081452.

[5] N. Ford, M. Richards, P. Sadalage, and Z. Dehghani, *Software Architecture: The Hard Parts Modern Trade-Off Analyses for Distributed Architectures*, 1st ed. O'Riley Media, 2021.

[6] V. Vernon and T. Jaskula, *Praise for Strategic Monoliths and Microservices*, 1st ed. Addison-Wesley Publishing, 2022.

[7] R. Su and X. Li, 'Modular Monolith: Is This the Trend in Software Architecture?', in *Proceedings of the 1st International Workshop on New Trends in Software Architecture*, in SATrends '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 10–13. doi: 10.1145/3643657.3643911.

[8] International Organization for Standardization, 'ISO/IEC 25010'. Accessed: Oct. 27, 2024. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[9] G. Pinto and F. Castor, 'Energy Efficiency: A New Concern for Application Software Developers Development of energy', *Commun ACM*, vol. 60, no. 12, 2017, doi: 10.1145/3154384.

[10] S. E. Sampson and M. J. Showalter, 'The performance-importance response function: Observations and implications', *Service Industries Journal*, vol. 19, no. 3, pp. 1–25, 1999, doi: 10.1080/02642069900000027.

[11] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice Fourth Edition*. 2021.

[12] N. Dragoni *et al.*, 'Microservices: yesterday, today, and tomorrow', in *Present And Ulterior Software Engineering*, Bertrand, Meyer, and M. Mazzara, Eds., 2017. Accessed: Oct. 27, 2024. [Online]. Available:

https://www.researchgate.net/publication/315664446_Microservices_yesterday_today_and_tomorrow

[13] B. Penzenstadler, V. Bauer, C. Calero, and X. Franch, 'Sustainability in software engineering: A systematic literature review', *IET Seminar Digest*, vol. 2012, no. 1, pp. 32–41, 2012, doi: 10.1049/IC.2012.0004.

[14] Ordem dos Engenheiros, 'Código de Ética e Deontologia'. Accessed: Nov. 13, 2024. [Online]. Available: https://www.ordemengenheiros.pt/fotos/editor2/regulamentos/codigo_ed.pdf

[15] D. Gotterbarn, K. Miller, and S. Rogerson, 'Software engineering code of ethics', *Commun ACM*, vol. 40, no. 11, Nov. 1997, doi: 10.1145/265684.265699/SUPPL_FILE/ACM-SE-CODE-OF-ETHICS.HTML.

[16] Instituto Politécnico do Porto, 'Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto', Nov. 12, 2020, *Porto*. Accessed: Nov. 13, 2024. [Online]. Available: https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf

[17] D. L. Parnas, 'On the criteria to be used in decomposing systems into modules', *Commun ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, doi: 10.1145/361598.361623.

[18] C. Lilienthal, 'Improve Your Architecture with the Modularity Maturity Index', in *Software Architecture Metrics*, 2022, ch. 4. Accessed: Oct. 31, 2024. [Online]. Available: https://www.oreilly.com/library/view/software-architecture-metrics/9781098112226/ch04.html

[19] Dr. H. Müller, 'Software Architecture Evaluation Methods and Tools: Analyzing methods and tools for evaluating software architectures to ensure adherence to quality attributes and design principles', *Distributed Learning and Broad Applications in Scientific Research*, vol. 6, pp. 1–14, Jun. 2020, Accessed: Dec. 24, 2024. [Online]. Available: https://dlabi.org/index.php/journal/article/view/16

[20] hello2morrow, 'Sonargraph Overview'. Accessed: Dec. 24, 2024. [Online]. Available: https://www.hello2morrow.com/products/sonargraph

[21] A. Caracciolo, M. Lungu, O. Truffer, K. Levitin, and O. Nierstrasz, 'Evaluating an Architecture Conformance Monitoring Solution', *Proceedings - 7th International Workshop on Empirical Software Engineering in Practice, IWESEP 2016*, pp. 41–44, May 2016, doi: 10.1109/IWESEP.2016.12.

[22] E. Whiting and S. Andrews, 'Drift and Erosion in Software Architecture: Summary and Prevention Strategies', *ACM International Conference Proceeding Series*, pp. 132–138, May 2020, doi: 10.1145/3404663.3404665.

[23] J. Thomas, A. Nicolaescu, and H. Lichter, 'Static and Dynamic Architecture Conformance Checking: A Systematic, Case Study-Based Analysis on Tradeoffs and Synergies', in *5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2017)*, 2017.

[24] F. A. Fontana, R. Roveda, and M. Zanoni, 'Technical Debt Indexes Provided by Tools: A Preliminary Discussion', *Proceedings - 2016 IEEE 8th International Workshop on Managing Technical Debt, MTD 2016*, pp. 28–31, Dec. 2016, doi: 10.1109/MTD.2016.11.

[25] P. C. Avgeriou *et al.*, 'An Overview and Comparison of Technical Debt Measurement Tools', *IEEE Softw*, vol. 38, no. 3, pp. 61–71, May 2021, doi: 10.1109/MS.2020.3024958.

[26] M. Barbacci, T. H. Longstaff, M. H. Klein, and C. B. Weinstock, 'Software Engineering Institute Quality Attributes', Dec. 1995. Accessed: Dec. 24, 2024. [Online]. Available: https://insights.sei.cmu.edu/library/quality-attributes/

[27] Grafana, 'Grafana k6 | Documentation'. Accessed: Dec. 24, 2024. [Online]. Available: https://grafana.com/docs/k6/latest/

[28] Grafana, 'Grafana k6 | Overview'. Accessed: Dec. 24, 2024. [Online]. Available: https://grafana.com/oss/k6/

[29] Apache, 'Apache JMeter'. Accessed: Dec. 24, 2024. [Online]. Available: https://jmeter.apache.org/index.html

[30] Cloud Native Computing Foundation, 'GitHub | Kepler'. Accessed: Dec. 24, 2024. [Online]. Available: https://github.com/sustainable-computing-io/kepler

[31] Cloud Native Computing Foundation, 'Exploring Kepler's potentials: unveiling cloud application power consumption | CNCF'. Accessed: Dec. 24, 2024. [Online]. Available: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/

[32] M. Villamizar *et al.*, 'Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures', *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 233–247, Jun. 2017, doi: 10.1007/S11761-017-0208-Y.

[33] O. Al-Debagy and P. Martinek, 'A Comparative Review of Microservices and Monolithic Architectures', *18th IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2018 - Proceedings*, pp. 149–154, Nov. 2018, doi: 10.1109/CINTI.2018.8928192.

[34] M. Villamizar *et al.*, 'Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud', *2015 10th Colombian Computing Conference, 10CCC 2015*, pp. 583–590, Nov. 2015, doi: 10.1109/COLUMBIANCC.2015.7333476.

[35] K. Gos and W. Zabierowski, ‘The Comparison of Microservice and Monolithic Architecture’, *International Conference on Perspective Technologies and Methods in MEMS Design*, pp. 150–153, 2020, doi: 10.1109/MEMSTECH49584.2020.9109514.

[36] M. A. Said, L. Belouaddane, S. Mihi, and A. Ezzati, ‘Modulith Architecture: Adoption Patterns, Challenges, and Emerging Trends’, *International Journal of Computing and Digital Systems*, Apr. 2024, doi: 10.12785/ijcds/XXXXXX.

[37] O. Drotbohm, ‘Introducing Spring Modulith’. Accessed: Oct. 27, 2024. [Online]. Available: https://spring.io/blog/2022/10/21/introducing-spring-modulith

[38] S. Petrovic and G. Sawhney, ‘Introducing Service Weaver: A Framework for Writing Distributed Applications’. Accessed: Dec. 22, 2024. [Online]. Available: https://opensource.googleblog.com/2023/03/introducing-service-weaver-framework-for-writing-distributed-applications.html

[39] R. Grandl, ‘A Quick Introduction to Service Weaver’. Accessed: Dec. 22, 2024. [Online]. Available: https://serviceweaver.dev/blog/quick_intro.html

[40] Google, ‘Service Weaver’. Accessed: Dec. 22, 2024. [Online]. Available: https://serviceweaver.dev/

[41] Service Weaber, ‘GitHub | Service Weaver - README.md’. Accessed: Dec. 28, 2024. [Online]. Available: https://github.com/ServiceWeaver/weaver/blob/main/README.md

[42] NetwrokNT, ‘Light | Light Hybrid 4j’. Accessed: Dec. 22, 2024. [Online]. Available: https://www.networknt.com/getting-started/light-hybrid-4j/

[43] NetworkNT, ‘Light | Hybrid Serverless Modularized Monolithic’. Accessed: Dec. 22, 2024. [Online]. Available: https://www.networknt.com/style/light-hybrid-4j/

[44] M. Fowler, K. Beck, J. Brant, W. Opdyke, and R. Don, *Refactoring: improving the design of existing code*. Addison-Wesley, Addison Wesley longman, Inc., 1999.

[45] A. Yamashita and L. Moonen, ‘Do code smells reflect important maintainability aspects?’, *IEEE International Conference on Software Maintenance, ICSM*, pp. 306–315, 2012, doi: 10.1109/ICSM.2012.6405287.

[46] F. Tian, P. Liang, and M. A. Babar, ‘How developers discuss architecture smells? An exploratory study on stack overflow’, *Proceedings - 2019 IEEE International Conference on Software Architecture, ICSA 2019*, pp. 91–100, Apr. 2019, doi: 10.1109/ICSA.2019.00018.

[47] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, ‘Toward a catalogue of architectural bad smells’, *Lecture Notes in Computer Science (including subseries Lecture*

Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 5581 LNCS, pp. 146–162, 2009, doi: 10.1007/978-3-642-02351-4_10.

[48] R. Laigner *et al.*, 'From a Monolithic Big Data System to a Microservices Event-Driven Architecture', *Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020*, pp. 213–220, Aug. 2020, doi: 10.1109/SEAA51224.2020.00045.

[49] M. Ezzeddine, S. Tauvel, F. Baude, F. Huet, and A. En Provence, 'On The Design of SLA-Aware and Cost-Efficient Event Driven Microservices', 2021, doi: 10.1145/3493649.3493657.

[50] H. Cabane and K. Farias, 'On the impact of event-driven architecture on performance: An exploratory study', *Future Generation Computer Systems*, vol. 153, pp. 52–69, 2024, doi: 10.1016/j.future.2023.10.021.

[51] B. M. Michelson, 'Event-Driven Architecture Overview', *Patricia Seybold Group*, vol. 2, pp. 10–1571, 2006, doi: 10.1571/BDA2-2-06CC.

[52] J. McGovern, O. Sims, A. Jain, and M. Little, 'Event-driven architecture', *Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations*, pp. 317–355, 2006.

[53] A. Bellemare, *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. Sebastopol, CA: O'Reilly Media, 2020.

[54] N. Garg, *Apache Kafka*. Packt, 2013. Accessed: Mar. 19, 2025. [Online]. Available: www.it-ebooks.info

[55] Apache, 'Apache Kafka'. Accessed: Mar. 19, 2025. [Online]. Available: https://kafka.apache.org/intro

[56] B. Stopford, *Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka*. Sebastopol, CA: O'Reilly Media, 2018.

[57] V. M. Ionescu, 'The analysis of the performance of RabbitMQ and ActiveMQ', *2015 14th RoEduNet International Conference - Networking in Education and Research, RoEduNet NER 2015 - Proceedings*, pp. 132–137, Oct. 2015, doi: 10.1109/ROEDUNET.2015.7311982.

[58] S. T and S. N. K, 'A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming', Dec. 2019, Accessed: Mar. 21, 2025. [Online]. Available: https://arxiv.org/abs/1912.03715v1

[59] VMWare Tanzu, 'RabbitMQ: One broker to queue them all | RabbitMQ'. Accessed: Mar. 21, 2025. [Online]. Available: https://www.rabbitmq.com/

[60] G. M. Roy, *RabbitMQ in Depth*. Shelter Island, NY: Manning Publications, 2018.

[61] M. Kuhrmann, D. M. Fernández, and M. Daneva, 'On the pragmatic design of literature studies in software engineering: an experience-based guideline', *Empir Softw Eng*, vol. 22, no. 6, pp. 2852–2891, Dec. 2017, doi: 10.1007/S10664-016-9492-Y/TABLES/8.

[62] M. J. Page *et al.*, 'The PRISMA 2020 statement: an updated guideline for reporting systematic reviews', *BMJ*, vol. 372, Mar. 2021, doi: 10.1136/BMJ.N71.

[63] S. Bhattacharya, K. Gopinath, K. Rajamani, and M. Gupta, 'Software bloat and wasted joules: Is modularity a hurdle to green software?', *Computer (Long Beach Calif)*, vol. 44, no. 9, pp. 97–101, Sep. 2011, doi: 10.1109/MC.2011.293.

[64] R. Perez-Castillo and M. Piattini, 'Analyzing the harmful effect of god class refactoring on power consumption', *IEEE Softw*, vol. 31, no. 3, pp. 48–54, 2014, doi: 10.1109/MS.2014.23.

[65] G. Dhaka and P. Singh, 'An empirical investigation into code smell elimination sequences for energy efficient software', *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 0, pp. 349–352, Jul. 2016, doi: 10.1109/APSEC.2016.057.

[66] F. Arcelli Fontana, M. Camilli, D. Rendina, A. G. Taraboi, and C. Trubiani, 'Impact of Architectural Smells on Software Performance: an Exploratory Study', in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, in EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 22–31. doi: 10.1145/3593434.3593442.

[67] D. Gravanis, G. Kakarontzas, and V. Gerogiannis, 'You don't need a Microservices Architecture (yet): Monoliths may do the trick', in *Proceedings of the 2021 European Symposium on Software Engineering*, in ESSE '21. New York, NY, USA: Association for Computing Machinery, 2022, pp. 39–44. doi: 10.1145/3501774.3501780.

[68] D. Taibi, V. Lenarduzzi, and C. Pahl, 'Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation', *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, Sep. 2017, doi: 10.1109/MCC.2017.4250931.

[69] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, 'From Monolithic to Microservices: An Experience Report from the Banking Domain', IEEE Computer Society, Aug. 2017. doi: 10.13140/RG.2.2.34717.00482.

[70] H. E. Hayretci and F. B. Aydemir, 'A Multi Case Study on Legacy System Migration in the Banking Industry', in *Advanced Information Systems Engineering: 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28 – July 2, 2021, Proceedings*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 536–550. doi: 10.1007/978-3-030-79382-1_32.

[71] T. Colanzi *et al.*, 'Are we speaking the industry language? The practice and literature of modernizing legacy systems with microservices', in *Proceedings of the 15th Brazilian Symposium on Software Components, Architectures, and Reuse*, in SBCARS '21. New York,

NY, USA: Association for Computing Machinery, 2021, pp. 61–70. doi: 10.1145/3483899.3483904.

[72] A. Razzaq, 'A Systematic Review on Software Architectures for IoT Systems and Future Direction to the Adoption of Microservices Architecture', *SN Comput. Sci.*, vol. 1, no. 6, Oct. 2020, doi: 10.1007/s42979-020-00359-w.

[73] D. Faustino, N. Gonçalves, M. Portela, and A. Rito Silva, 'Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation', *Perform. Eval.*, vol. 164, no. C, Jul. 2024, doi: 10.1016/j.peva.2024.102411.

[74] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam, 'Modularization of a Large-Scale Business Application: A Case Study', *IEEE Softw*, vol. 26, no. 2, pp. 28–35, Mar. 2009, doi: 10.1109/MS.2009.42.

[75] VMWare Tanzu, 'Spring Petclinic | GitHub'. Accessed: May 03, 2025. [Online]. Available: https://github.com/spring-projects/spring-petclinic

[76] N. Marmeleiro, 'NunoMarmeleiro/spring-petclinic: Spring PetClinic application containing Non-modular Monolith, Modular Modulith and Microservices'. Accessed: Apr. 30, 2025. [Online]. Available: https://github.com/NunoMarmeleiro/spring-petclinic

[77] P. Ferreira, 'KirinDev/spring-petclinic-modulith: Spring PetClinic application using Spring Modulith'. Accessed: Feb. 21, 2025. [Online]. Available: https://github.com/KirinDev/spring-petclinic-modulith/tree/main

[78] Spring, 'spring-petclinic/spring-petclinic-microservices: Distributed version of Spring Petclinic built with Spring Cloud'. Accessed: Feb. 26, 2025. [Online]. Available: https://github.com/spring-petclinic/spring-petclinic-microservices

[79] P. Dobbelaere and K. Sheykh Esmaili, 'Industry Paper: Kafka versus RabbitMQ', vol. 12, 2017, doi: 10.1145/3093742.3093908.

[80] R. van. Solingen and Egon. Berghout, 'The goal/question/metric method : a practical guide for quality improvement of software development', p. 199, 1999.

[81] V. R. Basili, G. Caldiera, and H. D. Rombach, 'THE GOAL QUESTION METRIC APPROACH', 1994.

[82] A. von Zitzewitz, 'A Promising New Metric To Track Maintainability – hello2morrow – Empowering Software Craftsmanship'. Accessed: Apr. 19, 2025. [Online]. Available: https://blog.hello2morrow.com/2018/12/a-promising-new-metric-to-track-maintainability/

[83] A. Hindle, 'Green mining: a methodology of relating software change and configuration to power consumption', *Empir Softw Eng*, vol. 20, no. 2, pp. 374–409, Apr. 2015, doi: 10.1007/S10664-013-9276-6.

[84] N. vad der Hoeven, ‘Comparing k6 and JMeter for load testing’. Accessed: May 18, 2025. [Online]. Available: https://grafana.com/blog/2021/01/27/k6-vs-jmeter-comparison/

[85] F. Tapia, M. ángel Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, ‘From monolithic systems to microservices: A comparative study of performance’, *Applied Sciences (Switzerland)*, vol. 10, no. 17, Sep. 2020, doi: 10.3390/APP10175797.

[86] D. Namiot and M. Senps-Sneppe, ‘On Micro-services Architecture’, *International Journal of Open Information Technologies*, vol. 2, pp. 24–27, Sep. 2014.

9 Appendix A

This appendix shows all the pages of the exported document based on the Microsoft Project of this academic paper.

ID	Task Mode	WBS	Task Name	WBS Level	Duration	Start	Finish
1		**1**	**Dissertation Project**	**Project**	**230 days**	**Mon 16/09/24**	**Fri 01/08/25**
2		**1.1**	**Dissertation Preparation**	**Phase**	**104 days**	**Mon 16/09/24**	**Thu 06/02/25**
3		**1.1.1**	**State-of-the-Art**	**Deliverable**	**80 days**	**Mon 16/09/24**	**Fri 03/01/25**
4		1.1.1.1	Background	Deliverable	80 days	Mon 16/09/24	Fri 03/01/25
5		1.1.1.2	Research Questions	Deliverable	45 days	Mon 04/11/24	Fri 03/01/25
6		1.1.1.3	Discussion	Deliverable	25 days	Mon 02/12/24	Fri 03/01/25
7		**1.1.2**	**Planning**	**Deliverable**	**45 days**	**Mon 04/11/24**	**Fri 03/01/25**
8		1.1.2.1	Project Charter	Deliverable	45 days	Mon 04/11/24	Fri 03/01/25
9		1.1.2.2	WBS	Deliverable	45 days	Mon 04/11/24	Fri 03/01/25
10		1.1.2.3	Planning	Deliverable	45 days	Mon 04/11/24	Fri 03/01/25
11		1.1.3	Dissertation Preparation Report 0.1	Deliverable	30 days	Mon 04/11/24	Fri 13/12/24
12		1.1.4	Dissertation Preparation Report Validation	Task	6 days	Mon 16/12/24	Mon 23/12/24
13		1.1.5	Final Dissertation Preparation Report	Deliverable	6 days	Tue 24/12/24	Tue 31/12/24
14		1.1.6	Dissertation Preparation Presentation	Deliverable	3 days	Wed 01/01/25	Fri 03/01/25
15		**1.2**	**Experiment Preparation**	**Phase**	**25 days**	**Fri 07/02/25**	**Thu 13/03/25**
16		1.2.1	Define criteria for selecting an open-source non-modular application	Task	5 days	Fri 07/02/25	Thu 13/02/25
17		1.2.2	Select and analyze the chosen open-source application (non-modular monolithic)	Deliverable	10 days	Fri 14/02/25	Thu 27/02/25
18		1.2.3	Define the migration strategy for modularization	Task	5 days	Fri 28/02/25	Thu 06/03/25
19		1.2.4	Establish the evaluation metrics	Task	10 days	Fri 28/02/25	Thu 13/03/25

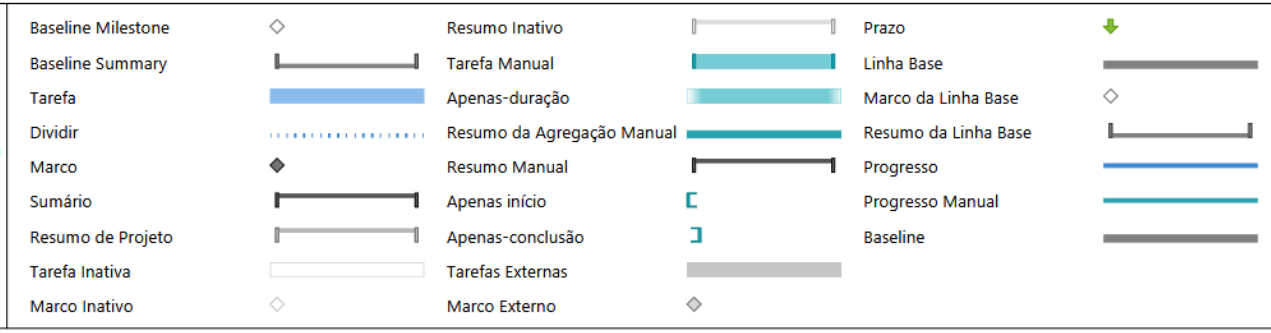
Projeto: Dissertation Project Pla
Data: Thu 26/12/24

Baseline Milestone	Resumo Inativo	Prazo
Baseline Summary	Tarefa Manual	Linha Base
Tarefa	Apenas-duração	Marco da Linha Base
Dividir	Resumo da Agregação Manual	Resumo da Linha Base
Marco	Resumo Manual	Progresso
Sumário	Apenas início	Progresso Manual
Resumo de Projeto	Apenas-conclusão	Baseline
Tarefa Inativa	Tarefas Externas	
Marco Inativo	Marco Externo	

Página 1

ID	Task Mode	WBS	Task Name	WBS Level	Duration	Start	Finish
20		1.2.5	Set up the experimental environment	Task	5 days	Fri 07/03/25	Thu 13/03/25
21		1.2.6	Sonargraph Improvement	Task	25 days	Fri 07/02/25	Thu 13/03/25
22		1.2.7	Grafana K6 Improvement	Task	25 days	Fri 07/02/25	Thu 13/03/25
23		1.2.8	Kepler Improvement	Task	25 days	Fri 07/02/25	Thu 13/03/25
24		**1.3**	**Migration**	**Phase**	**30 days**	**Fri 14/03/25**	**Thu 24/04/25**
25		**1.3.1**	**Modular Monolithic**	**Deliverable**	**30 days**	**Fri 14/03/25**	**Thu 24/04/25**
26		1.3.1.1	Module Boundaries	Task	10 days	Fri 14/03/25	Thu 27/03/25
27		1.3.1.2	Spring Modulith Improvement	Task	30 days	Fri 14/03/25	Thu 24/04/25
28		1.3.1.3	Code Refactor	Task	15 days	Fri 28/03/25	Thu 17/04/25
29		1.3.1.4	Functionality Validation	Task	5 days	Fri 18/04/25	Thu 24/04/25
30		**1.4**	**Evaluate**	**Phase**	**40 days**	**Fri 25/04/25**	**Thu 19/06/25**
31		1.4.1	Analysis	Deliverable	35 days	Fri 25/04/25	Thu 12/06/25
32		1.4.2	Report version 0.1	Deliverable	5 days	Fri 13/06/25	Thu 19/06/25
33		**1.5**	**Conclusions**	**Phase**	**23 days**	**Fri 20/06/25**	**Tue 22/07/25**
34		1.5.1	Final Report	Deliverable	10 days	Fri 20/06/25	Thu 03/07/25
35		1.5.2	Report Validation	Task	5 days	Fri 04/07/25	Thu 10/07/25
36		1.5.3	Scientific Report	Deliverable	5 days	Fri 11/07/25	Thu 17/07/25
37		1.5.4	Final presentation	Deliverable	5 days	Fri 04/07/25	Thu 10/07/25
38		1.5.5	Presentation Validation	Task	5 days	Fri 11/07/25	Thu 17/07/25
39		1.5.6	Presentation Preparation	Task	3 days	Fri 18/07/25	Tue 22/07/25

Projeto: Dissertation Project Pla
Data: Thu 26/12/24

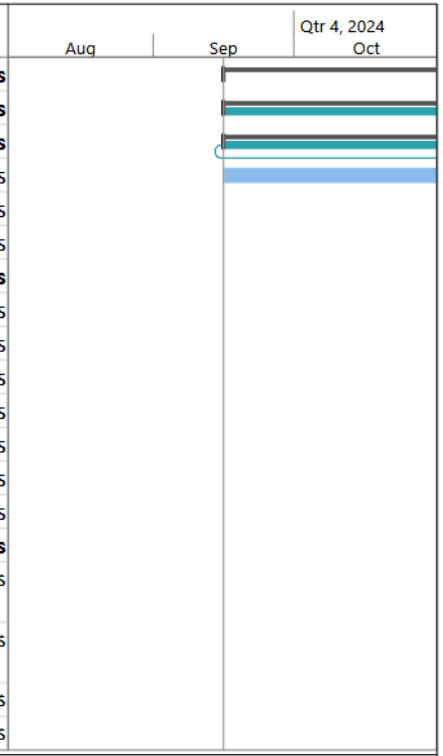


ID	Task Mode	WBS	Task Name	WBS Level	Duration	Start	Finish
40		**1.6**	**Milestones**	**Milestones**	**150 days**	**Fri 03/01/25**	**Fri 01/08/25**
41		1.6.1	State-of-art delivered	Milestones	0 days	Fri 03/01/25	Fri 03/01/25
42		1.6.2	Planning delivered	Milestones	0 days	Fri 03/01/25	Fri 03/01/25
43		1.6.3	Software finished	Milestones	0 days	Thu 24/04/25	Thu 24/04/25
44		1.6.4	Report 0.1 delivered	Milestones	0 days	Thu 19/06/25	Thu 19/06/25
45		1.6.5	Inte. Analysis finished	Milestones	0 days	Thu 19/06/25	Thu 19/06/25
46		1.6.6	Final report delivered	Milestones	0 days	Fri 25/07/25	Fri 25/07/25
47		1.6.7	Presentadion date	Milestones	0 days	Fri 01/08/25	Fri 01/08/25

Projeto: Dissertation Project Pla
Data: Thu 26/12/24

Baseline Milestone	Resumo Inativo	Prazo
Baseline Summary	Tarefa Manual	Linha Base
Tarefa	Apenas-duração	Marco da Linha Base
Dividir	Resumo da Agregação Manual	Resumo da Linha Base
Marco	Resumo Manual	Progresso
Sumário	Apenas início	Progresso Manual
Resumo de Projeto	Apenas-conclusão	Baseline
Tarefa Inativa	Tarefas Externas	
Marco Inativo	Marco Externo	

Predecessors	Resource Names	Cost	% Complete	Baseline Work
		100.00 €	**0%**	**1,064 hrs**
		0.00 €	**0%**	**360 hrs**
		0.00 €	**0%**	**0 hrs**
	Researcher	0.00 €	0%	0 hrs
	Researcher,supervisor	0.00 €	0%	0 hrs
5SS	Researcher	0.00 €	0%	0 hrs
		0.00 €	**0%**	**0 hrs**
	Researcher	0.00 €	0%	80 hrs
	Researcher	0.00 €	0%	40 hrs
	supervisor,Researcher	0.00 €	0%	240 hrs
7SF,3SF	Researcher	0.00 €	0%	0 hrs
11	supervisor	0.00 €	0%	0 hrs
12	Researcher	0.00 €	0%	0 hrs
13	Researcher	0.00 €	0%	0 hrs
2		**100.00 €**	**0%**	**160 hrs**
	Researcher,supervisor	0.00 €	0%	0 hrs
16	Researcher	50.00 €	0%	0 hrs
17	Researcher,supervisor	0.00 €	0%	0 hrs
17	Researcher,supervisor	50.00 €	0%	0 hrs



Projeto: Dissertation Project Pla
Data: Thu 26/12/24

Baseline Milestone	Resumo Inativo	Prazo
Baseline Summary	Tarefa Manual	Linha Base
Tarefa	Apenas-duração	Marco da Linha Base
Dividir	Resumo da Agregação Manual	Resumo da Linha Base
Marco	Resumo Manual	Progresso
Sumário	Apenas início	Progresso Manual
Resumo de Projeto	Apenas-conclusão	Baseline
Tarefa Inativa	Tarefas Externas	
Marco Inativo	Marco Externo	

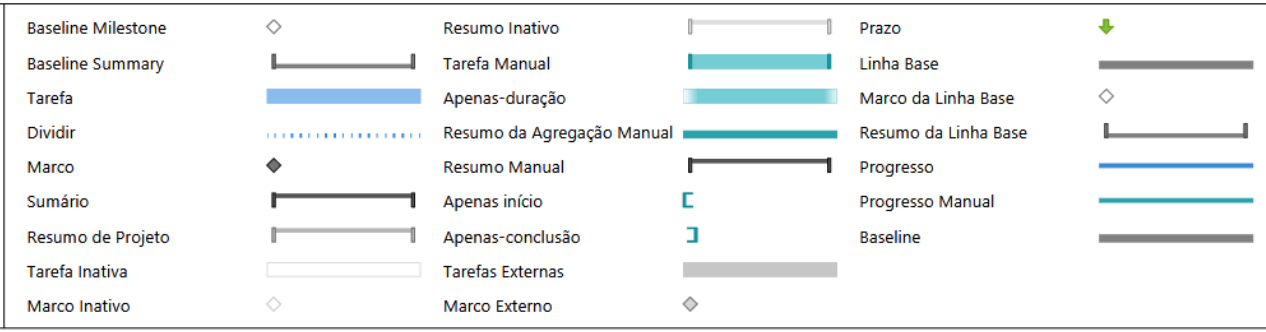
Predecessors	Resource Names	Cost	% Complete	Baseline Work	Aug	Sep	Qtr 4, 2024 Oct
16FF,17FF,18FF,19FF	Researcher	0.00 €	0%	0 hrs			
		0.00 €	0%	0 hrs			
		0.00 €	0%	0 hrs			
		0.00 €	0%	0 hrs			
15		**0.00 €**	**0%**	**144 hrs**			
		0.00 €	**0%**	**104 hrs**			
	Researcher	0.00 €	0%	40 hrs			
		0.00 €	0%	0 hrs			
26	Researcher	0.00 €	0%	40 hrs			
28	Researcher	0.00 €	0%	0 hrs			
24		**0.00 €**	**0%**	**200 hrs**			
	Researcher	0.00 €	0%	40 hrs			
31	Researcher	0.00 €	0%	80 hrs			
30		**0.00 €**	**0%**	**200 hrs**			
	Researcher	0.00 €	0%	80 hrs			
34	supervisor	0.00 €	0%	80 hrs			
35	Researcher	0.00 €	0%	0 hrs			
34	Researcher	0.00 €	0%	40 hrs			
37	supervisor	0.00 €	0%	0 hrs			
38	Researcher	0.00 €	0%	0 hrs			

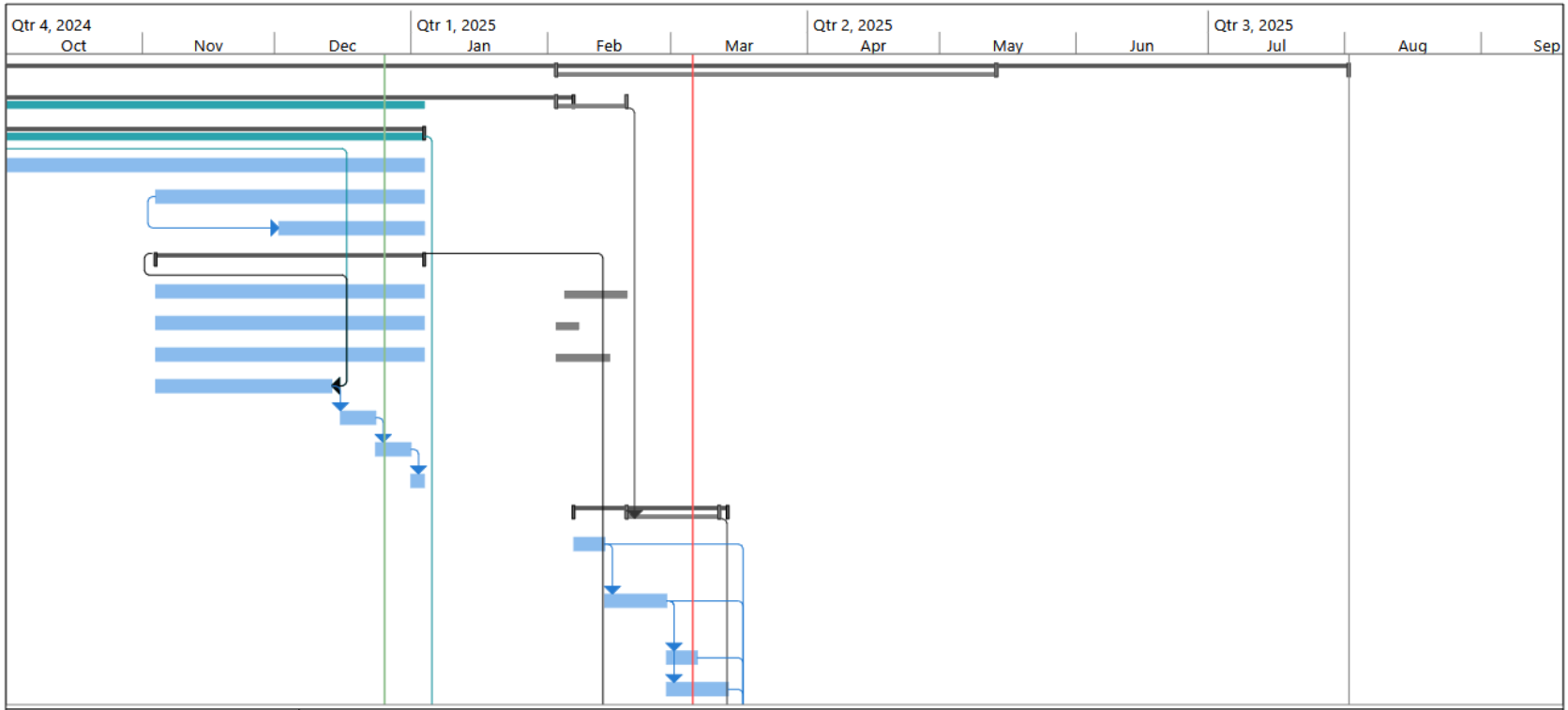
Projeto: Dissertation Project Pla
Data: Thu 26/12/24

Baseline Milestone	Resumo Inativo	Prazo
Baseline Summary	Tarefa Manual	Linha Base
Tarefa	Apenas-duração	Marco da Linha Base
Dividir	Resumo da Agregação Manual	Resumo da Linha Base
Marco	Resumo Manual	Progresso
Sumário	Apenas início	Progresso Manual
Resumo de Projeto	Apenas-conclusão	Baseline
Tarefa Inativa	Tarefas Externas	
Marco Inativo	Marco Externo	

Predecessors	Resource Names	Cost	% Complete	Baseline Work	Aug	Sep	Qtr 4, 2024 Oct
		0.00 €	**0%**	**0 hrs**			
3		0.00 €	0%	0 hrs			
7		0.00 €	0%	0 hrs			
24		0.00 €	0%	0 hrs			
32		0.00 €	0%	0 hrs			
32		0.00 €	0%	0 hrs			
35		0.00 €	0%	0 hrs			
39		0.00 €	0%	0 hrs			

Projeto: Dissertation Project Pla
Data: Thu 26/12/24

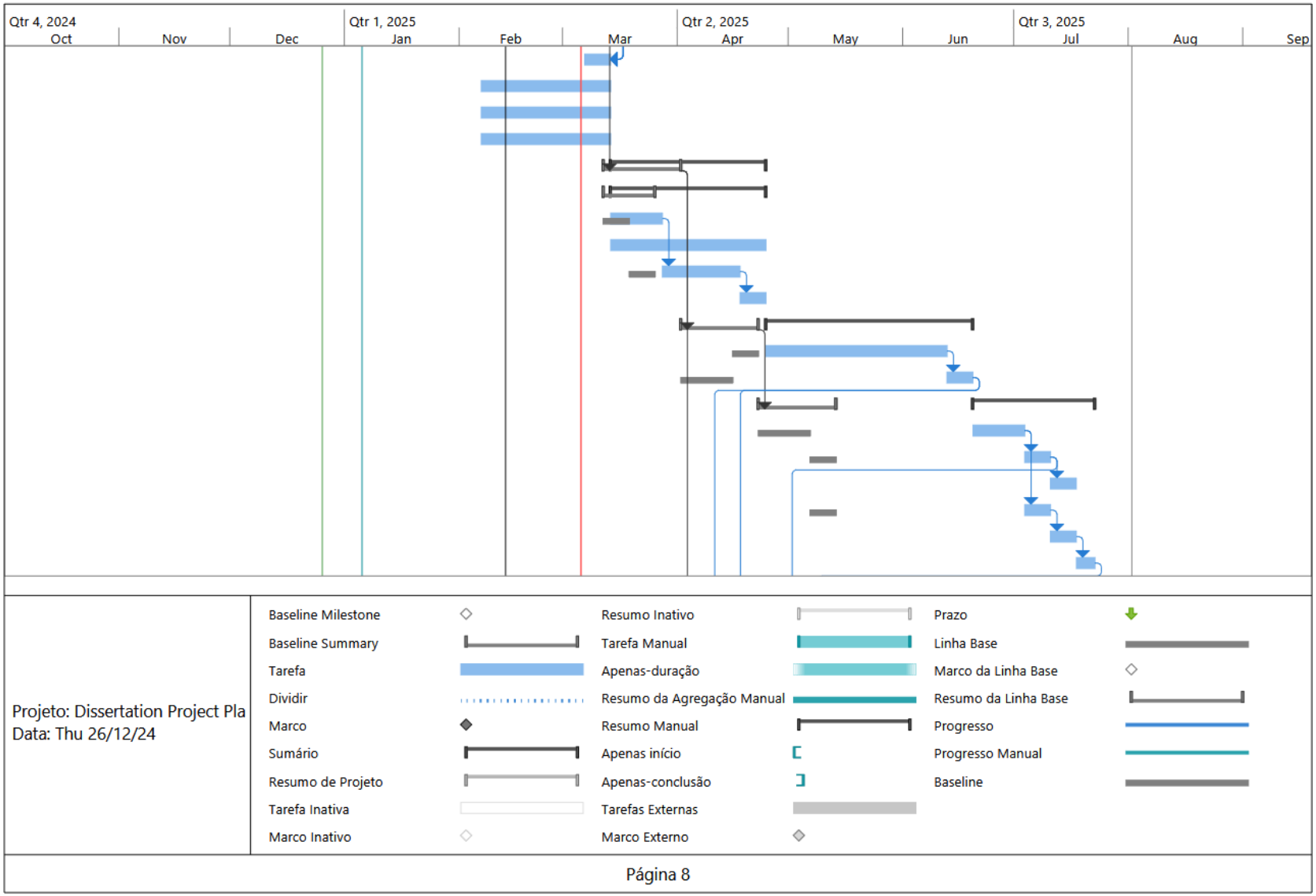


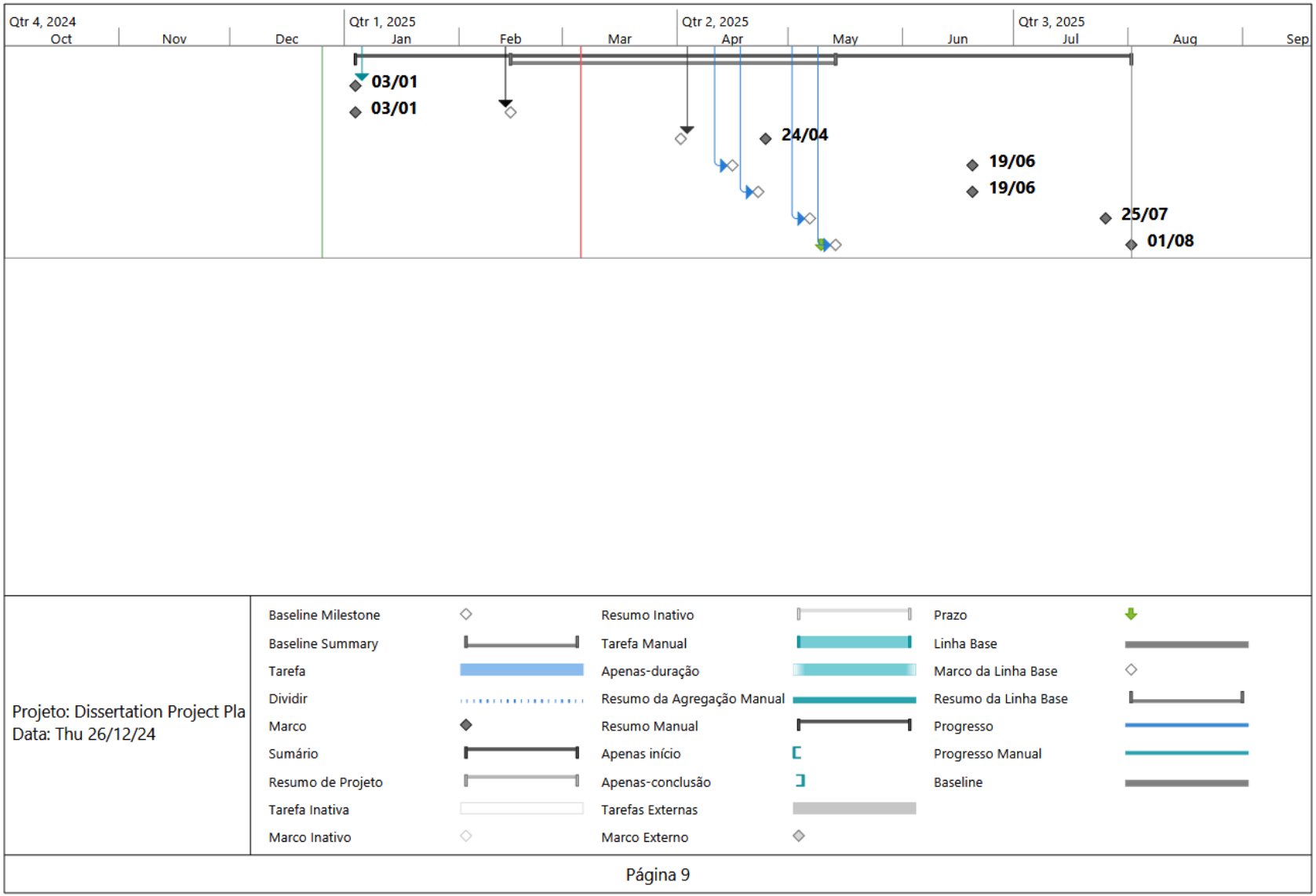


Projeto: Dissertation Project Pla
Data: Thu 26/12/24

Baseline Milestone	Resumo Inativo	Prazo
Baseline Summary	Tarefa Manual	Linha Base
Tarefa	Apenas-duração	Marco da Linha Base
Dividir	Resumo da Agregação Manual	Resumo da Linha Base
Marco	Resumo Manual	Progresso
Sumário	Apenas início	Progresso Manual
Resumo de Projeto	Apenas-conclusão	Baseline
Tarefa Inativa	Tarefas Externas	
Marco Inativo	Marco Externo	

Página 7


Qtr 4, 2024
Oct
Nov
Dec
Qtr 1, 2025
Jan
Feb
Mar
Qtr 2, 2025
Apr
May
Jun
Qtr 3, 2025
Jul
Aug
Sep
Projeto: Dissertation Project Pla
Data: Thu 26/12/24
Baseline Milestone
Baseline Summary
Tarefa
Dividir
Marco
Sumário
Resumo de Projeto
Tarefa Inativa
Marco Inativo
Resumo Inativo
Tarefa Manual
Apenas-duração
Resumo da Agregação Manual
Resumo Manual
Apenas início
Apenas-conclusão
Tarefas Externas
Marco Externo
Prazo
Linha Base
Marco da Linha Base
Resumo da Linha Base
Progresso
Progresso Manual
Baseline
Página 8


Qtr 4, 2024
Oct
Nov
Dec
Qtr 1, 2025
Jan
Feb
Mar
Qtr 2, 2025
Apr
May
Jun
Qtr 3, 2025
Jul
Aug
Sep
03/01
03/01
24/04
19/06
19/06
25/07
01/08
Projeto: Dissertation Project Pla
Data: Thu 26/12/24
Baseline Milestone
Baseline Summary
Tarefa
Dividir
Marco
Sumário
Resumo de Projeto
Tarefa Inativa
Marco Inativo
Resumo Inativo
Tarefa Manual
Apenas-duração
Resumo da Agregação Manual
Resumo Manual
Apenas início
Apenas-conclusão
Tarefas Externas
Marco Externo
Prazo
Linha Base
Marco da Linha Base
Resumo da Linha Base
Progresso
Progresso Manual
Baseline
Página 9

10 Appendix B

This appendix contains the *docker-compose.yml* file for the microservice application version of Spring Petclinic.

```
services:
  config-server:
    image: springcommunity/spring-petclinic-config-server
    container_name: config-server
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
    healthcheck:
      test: ["CMD", "curl", "-I", "http://config-server:8888"]
      interval: 5s
      timeout: 5s
      retries: 10
    ports:
     - 8888:8888

  discovery-server:
    image: springcommunity/spring-petclinic-discovery-server
    container_name: discovery-server
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
    healthcheck:
      test: ["CMD", "curl", "-f", "http://discovery-server:8761"]
      interval: 5s
      timeout: 3s
      retries: 10
    depends_on:
      config-server:
        condition: service_healthy
    ports:
     - 8761:8761

  ## Kafka
  kafka-server:
    image: apache/kafka:latest
    container_name: kafka-server
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_PROCESS_ROLES: broker,controller
      KAFKA_LISTENERS: PLAINTEXT://:9092,CONTROLLER://:9093
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka-server:9092
      KAFKA_CONTROLLER_LISTENER_NAMES: CONTROLLER
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
      KAFKA_CONTROLLER_QUORUM_VOTERS: 1@localhost:9093
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

```
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_NUM_PARTITIONS: 3
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
    ports:
      - "9092:9092"

  owners-service:
    image: springcommunity/spring-petclinic-owners-service
    container_name: owners-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
        reservations:
          cpus: 0.5
          memory: 256M
    depends_on:
      config-server:
        condition: service_healthy
      discovery-server:
        condition: service_healthy
    ports:
    - 8085:8085

  pets-service:
    image: springcommunity/spring-petclinic-pets-service
    container_name: pets-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
        reservations:
          cpus: 0.5
          memory: 256M
    depends_on:
      config-server:
        condition: service_healthy
      discovery-server:
        condition: service_healthy
    ports:
      - 8086:8086

  visits-service:
    image: springcommunity/spring-petclinic-visits-service
    container_name: visits-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
```

```
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
        reservations:
          cpus: 0.5
          memory: 256M
    depends_on:
      config-server:
        condition: service_healthy
      discovery-server:
        condition: service_healthy
    ports:
     - 8082:8082

  vets-service:
    image: springcommunity/spring-petclinic-vets-service
    container_name: vets-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
        reservations:
          cpus: 0.5
          memory: 256M
    depends_on:
      config-server:
        condition: service_healthy
      discovery-server:
        condition: service_healthy
    ports:
     - 8083:8083

  api-gateway:
    image: springcommunity/spring-petclinic-api-gateway
    container_name: api-gateway
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 512M
    depends_on:
      config-server:
        condition: service_healthy
      discovery-server:
        condition: service_healthy
    ports:
     - 8080:8080
```

11 Appendix C

This appendix represents the R script used for the Hypothesis Tests.

```
# ------------------------------
# Menu
# ------------------------------
# Define options
options <- c(
  "10 VUs - DELETE request",
  "100 VUs - DELETE request",
  "10 VUs - POST request",
  "100 VUs - POST request"
)

# Show menu
choice <- menu(options, title = "Select a load test configuration:")
# ------------------------------
# Load the Data
# ------------------------------

nonmodular <- switch(choice,
                read.csv("nonmodular-load-delete-10vus-treated.csv"),
                read.csv("nonmodular-load-delete-100vus-treated.csv"),
                read.csv("nonmodular-load-post-10vus-treated.csv"),
                read.csv("nonmodular-load-post-100vus-treated.csv")
)
modular <- switch(choice,
                      read.csv("modular-load-delete-10vus-treated.csv"),
                      read.csv("modular-load-delete-100vus-treated.csv"),
                      read.csv("modular-load-post-10vus-treated.csv"),
                      read.csv("modular-load-post-100vus-treated.csv")
)
microservices <- switch(choice,
                    read.csv("microservices-load-delete-10vus-treated.csv"),
                    read.csv("microservices-load-delete-100vus-treated.csv"),
                    read.csv("microservices-load-post-10vus-treated.csv"),
                    read.csv("microservices-load-post-100vus-treated.csv")
)
nm <- nonmodular$metric_value
m <- modular$metric_value
ms <- microservices$metric_value
# ------------------------------
# Normality Test (Lilliefors)
# ------------------------------
cat("\n--- Normality Test (Lilliefors) ---\n")
cat("NonModular:\n")
print(lillie.test(nm))
```

```r
cat("Modular:\n")
print(lillie.test(m))

cat("Microservices:\n")
print(lillie.test(ms))

# --------------------------------
# Skewness (Symmetry Check)
# --------------------------------

cat("\n--- Skewness (Symmetry Check) ---\n")
cat("NonModular Skewness:", skewness(nm), "\n")
cat("Modular Skewness:", skewness(m), "\n")
cat("Microservices Skewness:", skewness(ms), "\n")

# --------------------------------
# Summary Stats
# --------------------------------

cat("\n--- Summary Statistics ---\n")
summary_stats <- function(data, label) {
  cat("\n", label, "\n")
  print(summary(data))
  cat("Standard Deviation:", sd(data), "\n")
}

summary_stats(nm, "NonModular")
summary_stats(m, "Modular")
summary_stats(ms, "Microservices")

# --------------------------------
# Hypothesis Testing
# --------------------------------

# Helper to decide and run the right test
run_tests <- function(group1, group2, label1, label2) {
  cat(paste0("\n--- Comparing ", label1, " vs ", label2, " ---\n"))

  normal1 <- lillie.test(group1)$p.value >= 0.05
  normal2 <- lillie.test(group2)$p.value >= 0.05

  if (normal1 && normal2) {
    result <- t.test(group1, group2)
    cat("Using t-test (both normal)\n")
  } else {
    result <- wilcox.test(group1, group2)
    cat("Using Wilcoxon test (non-normal data)\n")
  }

  print(result)
}

# Pairwise comparisons
run_tests(nm, m, "NonModular", "Modular")
run_tests(nm, ms, "NonModular", "Microservices")
run_tests(m, ms, "Modular", "Microservices")
```

12 Appendix D

This appendix represents the results from the scenario with 10 concurrent users for the POST Request.

```
--- Normality Test (Lilliefors) ---
NonModular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  nm
D = 0.47748, p-value < 2.2e-16

Modular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  m
D = 0.47649, p-value < 2.2e-16

Microservices:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  ms
D = 0.44875, p-value < 2.2e-16

--- Skewness (Symmetry Check) ---
NonModular Skewness: 2.752873
Modular Skewness: 3.53797
Microservices Skewness: 5.118652

--- Summary Statistics ---
 NonModular
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5034   1.0623   1.5785   9.7861   1.7562 170.6607
Standard Deviation: 24.87709

 Modular
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.5029  1.0455  1.0663  6.6584  1.5794 94.4036
Standard Deviation: 20.26828

 Microservices
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.503   1.571   1.891   4.221   2.173 165.649
Standard Deviation: 11.20095

--- Comparing NonModular vs Modular ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 29284044, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing NonModular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 47140077, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing Modular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 41919095, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0
```

13 Appendix E

This appendix represents the results from the scenario with 10 concurrent users for the DELETE Request.

```
--- Normality Test (Lilliefors) ---
NonModular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  nm
D = 0.46779, p-value < 2.2e-16

Modular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  m
D = 0.47575, p-value < 2.2e-16

Microservices:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  ms
D = 0.45104, p-value < 2.2e-16

--- Skewness (Symmetry Check) ---
NonModular Skewness: 2.241136
Modular Skewness: 3.141574
Microservices Skewness: 5.073419

--- Summary Statistics ---
 NonModular
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5027   1.5846   2.0964  12.9933   2.6073 171.1710
Standard Deviation: 28.38557

 Modular
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.5029  1.0572  1.5695  7.9930  1.6436 96.3581
Standard Deviation: 22.08146

 Microservices
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5038   1.5681   1.6985   4.1762   2.1428 165.6776
Standard Deviation: 11.15204

--- Comparing NonModular vs Modular ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 32311882, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing NonModular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 77262972, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing Modular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 64258485, p-value < 2.2e-16
```

14 Appendix F

This appendix represents the results from the scenario with 100 concurrent users for the POST Request.

```
--- Normality Test (Lilliefors) ---
NonModular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  nm
D = 0.21807, p-value < 2.2e-16

Modular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  m
D = 0.28784, p-value < 2.2e-16

Microservices:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  ms
D = 0.24369, p-value < 2.2e-16

--- Skewness (Symmetry Check) ---
NonModular Skewness: 0.7478444
Modular Skewness: 0.5301057
Microservices Skewness: 0.879158

--- Summary Statistics ---
 NonModular
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5036   6.7346  90.4620  64.4630  98.7321 584.2935
Standard Deviation: 56.48834

 Modular
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5027   4.3851  10.9702  48.6212  93.4759 295.4389
Standard Deviation: 47.99075

 Microservices
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5139   7.7743  18.5699  46.0154  85.6672 320.3472
Standard Deviation: 43.65718

--- Comparing NonModular vs Modular ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 55650014, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing NonModular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 112295507, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing Modular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 127637437, p-value = 0.5227
alternative hypothesis: true location shift is not equal to 0
```

15 Appendix G

This appendix represents the results from the scenario with 100 concurrent users for the DELETE Request.

```
--- Normality Test (Lilliefors) ---
NonModular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  nm
D = 0.18069, p-value < 2.2e-16

Modular:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  m
D = 0.27848, p-value < 2.2e-16

Microservices:
    Lilliefors (Kolmogorov-Smirnov) normality test
data:  ms
D = 0.27195, p-value < 2.2e-16

--- Skewness (Symmetry Check) ---
NonModular Skewness: 0.878808
Modular Skewness: 0.4825097
Microservices Skewness: 1.080653

--- Summary Statistics ---
 NonModular
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5199   8.1966  92.9719  76.2497 101.1241 599.1006
Standard Deviation: 63.24843

 Modular
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5038   4.7713  14.1733  51.4115  94.6321 325.5361
Standard Deviation: 49.10034

 Microservices
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.5039   7.2056  14.8944  40.3805  82.4578 248.9584
Standard Deviation: 41.91504

--- Comparing NonModular vs Modular ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 58882998, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing NonModular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 126053383, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0

--- Comparing Modular vs Microservices ---
Using Wilcoxon test (non-normal data)
    Wilcoxon rank sum test with continuity correction
data:  group1 and group2
W = 138260765, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0
```